A Verified POSIX-Compliant Flash File System - Modular Verification Technology & Crash Tolerance

Gidon Ernst

Angaben zur Veröffentlichung / Publication details:

Ernst, Gidon. 2017. "A Verified POSIX-Compliant Flash File System - Modular Verification Technology & Crash Tolerance." Augsburg: Universität Augsburg.

A Verified POSIX-Compliant Flash File System

Modular Verification Technology & Crash Tolerance

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

Gidon Marian Ernst

Reviewers:	Prof. Dr. Wolfgang Reif Prof. Dr. Alexander Knapp Prof. Dr. John Derrick
Day of Defense:	November 28, 2016

Abstract

In the Flashix project, a file system for flash memory has been developed. It is proven functionally correct and tolerates system crashes such as abrupt power cuts at any point in time during its execution. The development approach is based in incremental and modular refinement. This dissertaion reports on the verification approach, the practical development, and the results.

The first contribution is a refinement theory with strong guarantees for compositionality built in. It is based on the observations that can be made of a sequential interface of subcomponents that have an encapsulated state. In order to be able to study the effect of power cuts, which may hit the system in any intermediate state of its execution, the foundations are a fine grained, trace-based semantics that exposes such steps. The integration of subcomponents with their context is done via operation calls with explicit input and output parameters. At such calls, the steps of the program defining the operation are collapsed into an atomic view, which provides the lever for a substitution theorem for submachine refinement.

The second contribution is an extension of the theory that permits to specify and verify the effect of system crashes and their subsequent recovery. This extension is fully compatible with the incremental and modular approach used for the functional verification. Furthermore, two different views of the atomicity of operations are considered, namely a white box semantics that is adequate for implementation level components, and a black box semantics that is adequate for specification level components. Several proof methods for compositional refinement in the presence of crashes are derived. A reduction theorem permits to gradually switch to the much simpler black box view.

The third contribution consists of formal models of concepts for flash file system that capture the implementation challenges at a high degree of abstraction, while at the same time these models do not introduce unrealistic conceptual simplifications. Specifically, a formal model of the POSIX standard for file systems is presented. Exploiting the modular theory for refinement under crashes, a verified implementation is described that separates generic aspects from the flash specific details. The models in the refinement hierarchy comprise a coherent working system, from which code is generated that can run on real flash hardware.

Acknowledgment

This thesis is part of the results of a long-term effort of quite a few people contributing in various ways to the success of the project.

I like to thank Prof. Dr. Wolfgang Reif for his continuous support and his critical remarks in favor of the more general perspective, complementing the technical side.

Many thanks to Prof. Dr. Alexander Knapp for a lot of interesting discussions that were always a very valuable source of inspiration and knowledge.

I am grateful to Dr. Gerhard Schellhorn for his patience to discuss even the most intricate details of flash file systems and refinement theory and for sharing his vast experience in modeling and proving.

Many thanks to all of my colleagues making the years interesting and fun. I had a great time working together with Jörg Pfähler, sharing the office and pushing the project towards completion. His work on on the parts not covered by this thesis and our cooperation on the overlap was awesome. I am happy to having shared the office with Bogdan Tofan as well, working together briefly on separation logic in the context of concurrency.

Of the many students who have participated in the project, I would like to especially thank Stefan Bodenmüller for continuously maintaining extending the models and proofs, Stefan Fritsch for realizing the tool support for the ASM theory, and Sarah Edenhofer and Jessica Tretter for pioneering several of the formal models in Flashix.

To my wonderful friends, my dad, my mam, my brother, and Martina—she's the best.

Contents

Chapter 1

Introduction

Summary. The topic of this thesis is the development of critical software systems to achieve to high assurance by the use of formal methods, which despite many recent advances is still a difficult and costly undertaking. This thesis contributes theoretical work on modular and incremental development of verified systems by *refinement* in the presence of *power cuts* that interrupt the regular flow of execution at any point in time. The theory is put to the test with the development of the first file system for flash memory that is proven functionally correct as well as power cut safe.

Contents

1.1 Software Development and Formal Methods

Background of this work is the prevalence and increasing complexity of embedded software systems. Some examples are the areas automotive (brakes, engine control, airbags), avionics (fly-by-wire), and aeronautics (flight control, communications, navigation, data storage). For the software deployed in these areas, high demands to the reliability and safety are typically made: the safety of people and the environment as well as large sums of money are at stake. Flaws in the design and implementation of file systems already lead to serious problems in mission-critical systems. A prominent example of a software defect is the Heartbleed bug in OpenSSL, effectively breaking the encryption of the communication between millions of internet users [48]. In 2004, the Mars Exploration Rover "Spirit" was almost lost during mission due to a severe file system bug [129].

Traditional approaches to improve the reliability of software include rigorous development processes, proper documentation of requirements, thorough testing, and code reviews. A complementary approach is to use mathematically founded techniques—*Formal Methods*—which are able to *prove the absence* of certain classes of software defects, thereby giving much higher confidence in the correctness of software than traditional techniques. Besides improving reliability, formal methods can reduce development costs, and lead to better documentation and maintainability of software. For these reasons, formal methods are becoming more popular in industry, see for example the survey [160].

As a practical example, Wheeler [156] provides a comprehensive analysis of Heartbleed

and discusses different approaches to formal analysis of software systems, their strengths and limits, and their potential to find such defects—or better—to prevent them beforehand. The quality assurance applied at NASA is described in [71]. Although a multitude of techniques are integrated, no full functional verification was done for the redevelopment of Spirit's successor "Curiosity", which promptly suffered from problems in 2013: A first incident turned out to be caused by physical corruption of one of the flash memory chips, but a second incident was traced to a software defect related to the file system implementation.[1] In 2016, Curiosity entered safe-mode again, supposedly due to a mismatch between the camera software and the data processing module.[2]

Widespread use of formal methods today is often limited to techniques that can be automated well, such as SAT solving and model checking. For data intensive systems with complex algorithms, automatic techniques cannot cope without human guidance and creativity, which is fairly difficult and requires expert knowledge.

Tony Hoare's *grand challenge for computing research* [84] calls for widespread collaboration to improve the verification technology that is available in order to to scale the use of formal methods to systems that are built and maintained in practice. The current trend trend is clear: in both academic and industrial contexts formal models are gradually applied to larger and more complex systems as demonstrated for example by the verified operating system seL4 [95]. Nevertheless, it remains active research area, as the goal has not yet been achieved to apply these techniques *routinely* to the development of complex software systems that are composed of many different components and span many conceptual layers of abstraction.

The incident with the Mars Rover Spirit prompted Joshi and Holzmann [91] from the NASA/JPL to propose the verification of a file system for flash memory as a pilot project for Tony Hoare's long-term grand challenge. This thesis takes up the challenge by the development of the Flashix file system, the first verified implementation of a file system for flash memory.

1.2 File Systems and the POSIX Standard

A file system provides the familiar structure of files, directories, and paths on top of a low-level representation that is mapped down to the bytes and blocks of the storage hardware. It offers operations to create/delete files and directories and to access their content and metadata. These operations are typically exposed to client applications through an operating system, which in turn fixes a stable interface on which application programmers can rely on.

Such an interface for the access to the file system is defined as part of the Portable Operating System Standard (POSIX) [3]. The standard covers many aspects including a description of the high-level view of a file system as well as the underlying data model. Specifically, POSIX file systems are hierarchical name spaces, where each entity is either a directory or a file. These are addressed by paths. The view of file content is that of a flat sequence of bytes, which is addressed indirectly through file handles that are acquired and released explicitly by an application. Metadata of file system entities comprises access rights, several time stamps (e.g. for the creation and most recent modifications), and various counters such as the number of entries within a directory.

[1] `http://www.jpl.nasa.gov/news/news.php?feature=3732`, article from March 18, 2013.
[2] `http://www.jpl.nasa.gov/news/news.php?feature=6559`, article from July 6, 2016.

POSIX knows several advanced concepts that optimize various access patterns in practice. For instance, shallow copies using hard-links (multiple names for one file) provide a convenient mechanism to backup large amounts of data incrementally without duplication. Furthermore, the POSIX standard specifies to some extend the degree of atomicity of operations regarding concurrent access and power cuts. Applications such as data bases rely on these features to provide higher-level abstractions on top of the POSIX interface.

The task of a file system is complex as it is burdened to realize the mentioned concepts. There are many different existing implementations with different design goals, trading for example between simplicity, performance, robustness, and features [102].

Modern operating systems such as Linux, Mac OS X, and different incarnations of BSD adhere (mostly) to the POSIX standard. Windows provides many of the necessary interfaces as part of the standard C library and with Cygwin[3] there is an emulation layer that bridges the remaining gaps. The bottom line is that POSIX is an *established* standard and therefore serves as the target specification of this thesis.

1.3 Flash Memory

Flash memory is a persistent mass storage technology. It is built into a large number of different products, such as USB drives and memory cards. Over the last years it has replaced traditional magnetic hard drives in personal computers in the form of Solid State Disks or hybrid disks where it is used as an intermediate cache, and it is becoming more widespread in the server market as well. The reason for the popularity of flash memory is that it has higher access speeds than traditional hard drives and consumes less power. It does not contain any moving parts and is therefore more shock-resistant and compact. For these reasons it is predominant in embedded systems that must endure extreme conditions, such as the onboard computers in space crafts and the already mentioned Mars exploration vehicles.

The downsides of flash memory are that it strongly restricts write access. The storage hardware is partitioned into *erase blocks*, each of which consists of a fixed number of physical memory pages. Write access is offered at the level of whole pages only, which must be written sequentially within each block. Storage space cannot be overwritten directly and can only be reclaimed for further use by erasing physical blocks in their entirety. Erasing is a relatively slow operation and has the effect that it wears out the memory cells over time. Modern hardware supports between 10^4 and 10^6 erase cycles before a block breaks down and becomes unusable. Flash memory is inherently unreliable: It may happen during ordinary operation that operations fail sporadically, prompting a subsequent erase cycle to recover the block for further use (in which case the present data should be backed up to a different location first).

Most consumer flash memory comes with a built-in controller that exposes a traditional interface to the operating system, specifically, they permit (virtual) overwriting of data in place. Such a controller is called a Flash Translation Layer (FTL). It has the advantage that existing file systems can be used without modifications to work on flash memory. Implementing efficient and correct FTLs is a difficult task. Early controllers in (cheap) USB pen drives for example were not spreading out erases evenly, quickly leading to many unusable blocks; this effect shows up as decreased overall capacity and ultimately leads to loss of data. Modern flash controllers addressing these problems are highly complex and

[3] https://www.cygwin.com

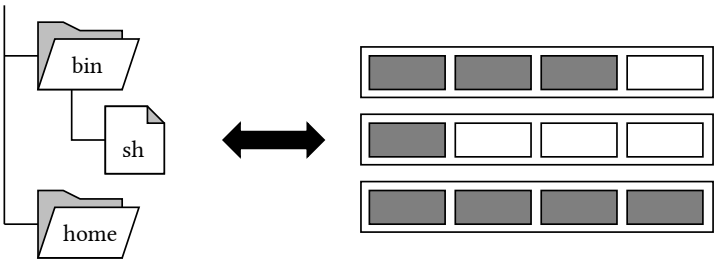


Figure 1.1: Comparison of the POSIX view of a file system as a hierarchy of directories and files (left) to the flash memory layout consisting of erase blocks and physical hardware pages. The conceptual gap in between the two worlds needs to be bridged by a flash file system.

have a long history of defects and bugs in the controlling firmware [153, 163].

An alternative to translation layers is the use of a Flash File System (FFS) that is specifically designed to work with the raw flash memory interface. While an FTL must employ a conservative strategy in order to present a hard-disk like interface, integrating decisions when to erase blocks and where to store data into the file system itself has the potential of being more efficient due to the additional information.

The challenge to design an efficient and robust FFS stems from balancing strong requirements of the interface exposed to applications and the weak guarantees offered by the hardware. Specifically, in high-assurance applications it is expected that power cuts and likewise the sporadic hardware errors do not lead to data loss. Accessing files and directories should take effect perceivably atomic, i.e., an operation succeeds completely or fails without modifying the observable state at all. As an example, the Unsorted Block Image File System (UBIFS) [69, 88] is a modern file system for flash memory that is part of Linux since 2008 and represents the state of the art.

1.4 Research Challenges

A study by Lu et al. [102] examines the reliability of existing file system implementations in Linux. The authors observe that "semantic bugs [. . .] are the dominant bug category" (Section 1) and suggest the use of formal methods, because these types of bugs are particularly hard to find using traditional approaches such as testing. Why is that so?

First, the gap between the high-level tree-based view and the low-level byte-based representation of the hardware is quite high, as already outlined in the previous sections and visualized for flash memory in Figure 1.1. The strategies to provide reliable and efficient data storage services are complex, especially for flash memory (see e.g. the UBIFS whitepaper [88]). These strategies must then be realized correctly in the implementation. To do so, the file system code must interact with the operating system's infrastructure as well as the driver level. These interfaces come with more or less well documented functionality, assumptions, and with varying ease of use. For instance, despite the existence of the thorough textual specification of the POSIX system interface [3], there are still semantic ambiguities as pointed out in [132], and the guarantees under power cuts are even less clear [125]. Hence, there are many potential sources of errors, even if shallow programming bugs associated with the low-level nature of the commonly used C language are not counted in.

The point made for the use of formal methods in [102] aims at reducing the high-level—semantic—problems related to 1) implementation concepts on one hand and 2) the integration with existing infrastructure on the other hand. Incremental, stepwise, and modular formal modeling of a system addresses the first point, whereas a precise, mathematical specification of the system's boundary can clarify the interaction, or at least properly document the assumptions made.

However, as indicated at the beginning of this chapter, use of formal methods still no routine activity, which is emphasized by the roadmap for NASA's flash file system challenge by Freitas et al. [60], too. Two aspects of Tony Hoare's grand vision [84] are singled out here that complement the previous arguments, providing a broader context for the research and goals (and results) of this thesis:

Research efforts should challenge the state of the art. Development of new and improved theories for specifying systems and verifying properties makes formal methods applicable to novel problem domains in the first place.

A modern and efficient flash file system incorporates many data structures and algorithms related to the mapping of the high-level directory structure down to the bytes and blocks of the hardware. These can be attributed to different conceptual views and consequentially the associated implementation and verification problems should be addressed individually and preferably abstracted within separate components. Therefore, a *modular* and *incremental* approach is required.

On the other hand, power cut tolerance is a concern that pervades the entire system. The strategies necessary to deal with this issue involve close cooperation of different parts of the system. The mathematical theory behind the corresponding proofs has started to be developed in the past few years only and is still fairly unexplored.

The main challenge therefore lies not only in expressing in the first place what it means for a system to be power cut safe but more importantly to develop a verification methodology that integrates well with the modular decomposition and conventional proofs for functional correctness.

Methods should be shown to be effective. Conducting actual case studies to evaluate and fine-tune formal theories contributes experience about best practices, effort, and costs; and it is a prerequisite for scaling to realistic systems and problem sizes.

The flash file system project is large enough that it is no longer sufficient that a chosen approach and mathematical theory is applicable in principle but that it copes in *practice* as well. Many concepts that have been described only informally so far are given a precise and understandable meaning in terms of formal models. The final goal is to end up with running code that can be integrated into the existing software landscape, one cannot afford to make unrealistic assumptions or to expose interfaces that are impractical.

As a result one has to deal with many issues that tend to be neglected in many formal developments that regard specific aspects in isolation only. With the integration of these aspects into a coherent system it shows whether the methodology is effective in the sense that it can solve real-world problems while at the same time the effort to conduct the development and proofs does not hit systematic limitations.

1.5 Approach and Methodology

Flash File System Concepts. This thesis relies on established standards and modern concepts in order to provide a flash file system. The requirements are expressed by an abstract formal model of the POSIX standard [3]. POSIX is a textual specification of operating system services and interfaces, including an extensive documentation of conformant file system behavior, defining names of operations and their parameters.

The implementation concepts are taken from existing solutions. For the generic part that is independent of the specifics of flash memory, the design is inspired by the Virtual Filesystem Switch architecture of Linux. For the flash specific data structures and algorithms, UBIFS [88] is taken as a blueprint as well as its submodule UBI [69] which is an abstraction layer of the hardware interface.

Hierarchical Components. Underlying this work is a hierarchical model of systems expressed by *components*. These are integrated via *interfaces*, which expose operations that can be called from the outside. To illustrate a system's architecture and likewise the refinement relations between models of different abstraction, diagrams of two types are used. Borrowing notation from the Unified Modeling Language (UML) [4], Figure 1.2 shows two components that are integrated via an interface that is expressed by the symbol —◎—. Diagrams like Figure 1.2 serve as the graphical representation of the first of the two structuring mechanisms that underlie the formal specification and analysis in this thesis.

Technically, components in this work are represented by a specific class of Abstract State Machines (ASMs) [29, 72] that provide an explicit representation and modeling support for the different types of diagrams shown here.

Observational Refinement. The methodological basis for verification is *correctness by construction*: One starts with an abstract, mathematically precise model of the system's behavior, which specifies exactly *what* the system should do, but now *how* this should be done. In a series of *refinement* steps, this abstract description of the system is transformed into the final software with full details concerning algorithms and data structures. A proof accompanies each step stating that the refined, more concrete model of the system exhibits the same behavior as the abstract one—the proof *verifies* the correctness of the final software product. The idea of correctness by construction in software engineering has a long history and comes in many different flavors, see for example [13, 14, 44, 83, 101, 157].

Here, refinement will be based on *observable equivalence*, captured in terms of the inputs and outputs made by the system as its computation progresses through time, which complements the static composition of the system described so far by its dynamics.

Graphically, refinement of a specification towards its implementation is denoted by dashed lines as shown in Figure 1.3. The specification (white) is connected to the corresponding implementation (grey) by a formal proof that establishes their observational equivalence.

Modularity. Figure 1.4 shows the static composition of a system with a *hierarchy* of components in three levels of abstraction, combining the two concepts of interface composition and refinement. A top-level specification is decomposed into some parts that can be made concrete already and a subcomponent that is to be refined further. On the intermediate level, an interface —◎— is established. In the next chapter, Figure 2.2 shows the structure of the whole Flashix file system as such a diagram.

From the engineering perspective, the benefit of this scheme is that the subcomponent

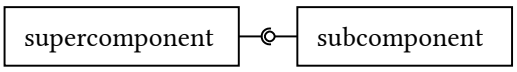


Figure 1.2: Component composition.

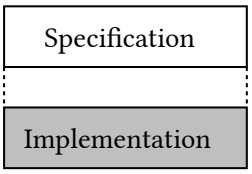


Figure 1.3: Refinement.

can be developed further without considering the part of the implementation split off at this stage while there is a formal guarantee that combining all the implementations (grey) produces a system that adheres to the topmost specification. The huge plus of this approach is that one can decompose not only the problem domain on a conceptual level, but structure the verification just the same, breaking the effort into individual pieces with a complexity that becomes manageable.

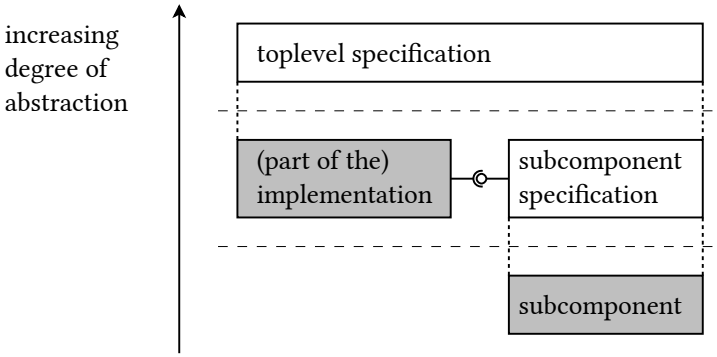


Figure 1.4: Modular refinement of a hierarchical system.

Crash Tolerance. A power cut is the sudden loss of power supply with the consequence that the system stops computing and communicating immediately. Power cuts and in general system *crashes* are problematic in particular related to data storage: a running write operation may be interrupted in the middle, leading to half written and possibly garbage data, depending on the physical characteristics of the storage medium. Not all operations can easily be implemented atomically, for example when several blocks are affected by one operation. The fundamental problem is to reconstruct from partial data without further knowledge a view consistent with the effect of the operation aborted at the time of the crash: Either the effect of the operation has taken place entirely or not at all.

While the implementation level strategies to mitigate the effect of power-cuts are quite mature (see e.g. [135]), formal approaches that permit to specify and verify desirable properties of a system under power cuts are being explored only recently. The theory behind such undertakings must necessarily be able to reason about *intermediate* steps of a computation, which is in stark contrast to many existing approaches to program verification that are based on pre- and postconditions [44, 82].

The subject is addressed here by a fine-grained program semantics that exhibits the necessary intermediate steps, in conjunction with a mechanism to increase the degree of atomicity wrt. when crashes happen to simplify the verification. The theory is seamlessly integrated with refinement, to yield a uniform and convenient approach that covers the whole verification effort.

1.6 Contributions of this Thesis

This thesis contributes to the advancement of system's development using formal methods in the two regards as outlined in Section 1.4. The contribution is threefold: On the theoretical side, a novel modularity theorem for refinement and a comprehensive specification and proof methodology for power cut analysis are presented. Development of the case study using this theory contributes the practical aspect.

Modular Refinement.

- A component model for hierarchical systems is developed, where components are represented by "data type like" Abstract State Machines with an interface defined by operations and explicitly declared subcomponents, called submachines. The defining feature of the theory is that it aligns a fine granular semantics of programs to the execution traces of the whole system. The dual purpose of execution traces permits to relate the steps of a context program to the operation calls to the submachine.
- A compositionality result will shown that explicitly recording input/output traces of systems is a sufficient criterion for substitutivity in sequential contexts that are again data type like ASMs. This contrasts existing proofs in the literature for e.g. data refinement [41, 64], which are based on the stronger criterion of simulation as a proof method, and for action systems [14], which are based on parallel composition.

Crash-Safety.

- A framework for the specification of the effect of power cuts and recovery is introduced. The theory is integrated into the refinement approach and gives a precise account of the behavior of the corresponding system. Sufficient syntactic proof obligations in temporal logic are given. It is shown that the modularity theorem proved for non power cut aware systems propagates to the extension.
- Taking the dual role of traces as the high-level steps of a system and the small steps of programs provides a lever to systematically switch the degree of atomicity under which power cuts are analyzed. The central result is a reduction theorem that permits to decrease the verification effort in practice significantly by giving sufficient conditions when atomicity can be increased. Specifically, it is shown how to forgo with temporal logic proof obligations in favor of conventional pre-/post verification.

The Flashix file system.

- The work in the context of the Flashix file system has produced formal models that capture a number of flash-relevant implementation concepts. The need to state these clearly to make the development amenable to formal proofs implies that these models can illustrate and document the internals of modern flash storage systems, not only *how* these work, but also *why* the implementation strategies are correct. As this thesis covers a large part of the system, it contributes general insights to the development of such systems.
- A significant outcome of the whole project is a working prototype that can run on existing hardware and can be integrated into today's software landscape. The Flashix file system is therefore an alternative to present solutions when high-assurance guarantees are needed.
- Finally, by the effort of the whole Flashix-team, NASA's challenge [91] is solved.

Outline

Chapter 2 gives an overview of the Flashix project and its goals and discusses the correctness guarantees and assumptions as well as the use of the file system in practice. Challenges and approaches to address these are outlined alongside an description of the different components of the system.

Chapter 3 provides some background on algebraic specifications, the ASM formalism and refinement, the KIV verification system, as well as separation logic, which is used in some of the refinement proofs.

Chapter 4 gives an account of data type like ASMs and the sequential programming language that defines their operations in terms of traces, called intervals here. Intervals are used to represent runs of transition systems, which give meaning to machines, and to represent the execution steps of programs as well.

Chapter 5 demonstrates that this dual use leads to a composition theorem that permits to nest refinements as shown in Figure 1.4.

Chapter 6 complements Chapter 2 with a technical overview of the refinement hierarchy, briefly sketching the models to give a coherent picture. The Chapters 7 to 10 detail the formal models and refinements of the flash file system development. The challenges outlined in Chapter 2 are realized with the system model of Chapter 4, and the ideas behind the respective formalizations are presented.

Chapter 11 introduces power cuts and crashes in general into theory as events that disrupt the normal flow of execution of the system. Focus lies on a) explicit modeling support for the concepts involved such as specification of the effect of power cuts and integration of recovery operations and b) approaches to reduce the verification effort by exploiting a connection between power cuts and error handling.

Chapter 12 complements the functional aspects of Chapter 7 to 10 by giving details how exactly crash safety is realized for Flashix and how the collaborative effort of recovery after such an event can be achieved in the first place: it will be shown that crashes lead to subtle effects in the behavior of the system and it is often not clear how to address these correctly.

The results and statistics are summarized in Chapter 13: what were the challenges encountered in practice to solve the large case study? What is the influence of the theory and also the tool support?

Chapter 14 draws final conclusions and outlines future work.

Appendix A contains a systematic overview of the models, outlining the concepts solved, their representation of state, and their interface.

Publications

Parts of the contributions presented in this thesis are based on previous publications:

Articles

1. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming (SCP)*, 2016. In Print.
2. 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.

In Proceedings

1. 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.
2. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper.
3. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular refinement for submachines of ASMs. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 188–203. Springer, 2014.
4. 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.
5. 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 Hardware and Software: Verification and Testing (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
6. 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. Elsevier, 2012.

Chapter 2

The Flashix File System

Flash, a-ah, savior of the universe — Queen

Summary. Flashix is a POSIX compliant file system for flash memory that can be used in a real world context. This chapter outlines the scope of the project, the chosen approach to its design, the system's architecture and implementation concepts. We start from an intuitive, top-level specification in terms of a tree of directories and files. It is explained how the gap down to the erase blocks and bytes of the flash hardware is bridged. Along the way, challenges and solutions to problems specific to the characteristics of flash memory are outlined.

Publications: This chapter is based on [144].

Contents

Goal of the Flashix project is to develop a file system for raw flash memory according to high assurance standards. The project follows the *correctness by construction* approach: An abstract but precise, formal specification of the requirements is gradually and stepwise transformed towards an implementation with all algorithmic details.

The file system should be *realistic*, which means two things specifically: Its interfaces to the outside world adhere to established standards, namely the Portable Operating System Standard (POSIX) and the Memory Technology Device (MTD) interface, so that it can be integrated into existing systems without trouble. State of the art implementation strategies are employed to deal with the specifics of flash memory in order to guarantee that there are a priori efficiency limitations built in. For the latter, the design of Flashix is based on the file system architecture of Linux, the existing Unsorted Block Image Sile System (UBIFS) [88] for flash memory, and its erase block management layer UBI [69].

2.1 High-Level Description

The Flashix file system consists of several parts as shown in Figure 2.1 that mirror this approach: A top-level specification of the POSIX file system interface defines the interface exported to the application layer. Flashix provides the operations `create`, `mkdir`, `rmdir`, `link`, `unlink`, `rename`, `open`, `close`, `read`, `write`, and `truncate` as described by the textual

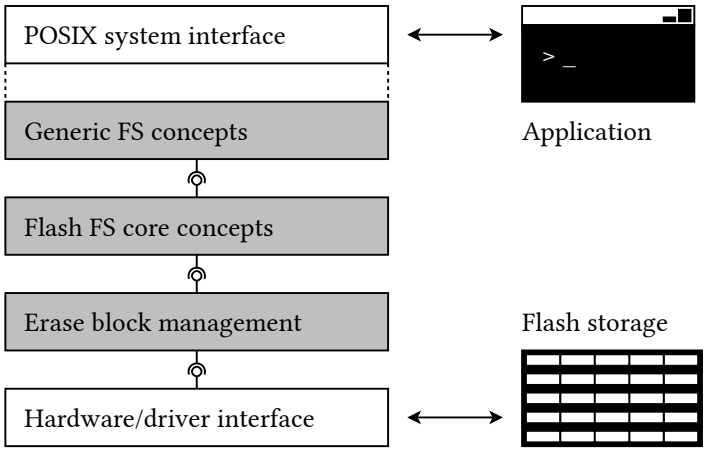


Figure 2.1: High-level structure of the Flashix file system showing the system's boundary interfaces at the top/bottom and the main conceptual components. For the components marked in grey an implementation is provided, whereas the white ones are specified abstractly and integrate Flashix into existing infrastructure.

standard in [3], respectively by the manual pages in section 2 that can be viewed with the `man` command line tool on a UNIX system. With slight deviations from the standard, Flashix provides `readdir` (the low-level/system interface is unspecified by POSIX), `readmeta` (corresponding to the `stat` system call), and `writemeta` (subsuming `chmod`/`chown` etc).

The implementation of the POSIX operations is split into three major components marked grey in Figure 2.1, in software engineering terms it is a three layer architecture. A generic part that realizes file system concepts such as path resolution and checking of access rights, a flash specific core that deals with high-level data structures and algorithms related to the storage hardware, and an abstraction layer that mediates access to the hardware in terms of logical erase blocks. These three parts can be used independently of each other—it is possible to integrate them with other developments such as [93].

The flash storage is accessed through a driver interface that exposes the low-level operations `read`, `write`, and `erase`, as well as bad block management via two operations `is_bad` and `mark_bad` at the level of physical erase blocks. This interface is modeled after the Memory Technology Device (MTD) interface of Linux [69] and encodes the assumptions made about the behavior of the hardware. It has a critical role in the context of the whole development as the overall correctness of the file system *in practice* depends on the adequacy of these assumptions.

The Flashix file system encompasses the complete stack of layers of Figure 2.1. These are further subdivided into different subcomponents, each one with a formal specification and a verified implementation as explained in the following section.

The (formal) development focuses on the *conceptual challenges* of building an FFS, which means that algorithms and data structures, as well as the architectural decomposition and invariants necessary for the verification are of particular interest. For this reason, the executable code is generated automatically, in contrast to a verification directly on the code level.

In the remainder of this chapter, the Flashix file system is reviewed from several perspectives. Section 2.2 summarizes the main concepts found in modern flash file systems

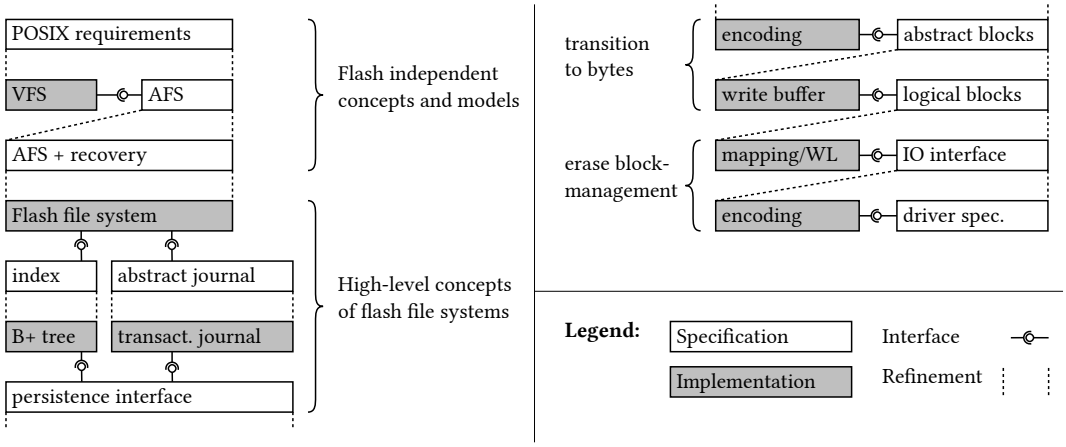


Figure 2.2: Detailed component structure of the system. The white boxes represent abstract specification models. The grey boxes represent concrete implementation models from which the final code is composed. Refinements are indicated by dotted lines.

and explains how these are realized (modularly) as part of the architecture shown in Figure 2.1. Section 2.3 addresses the view of the proof engineer: What are the guarantees that the system provides? What are the assumptions about the behavior of the hardware under normal operations, hardware errors, or power cuts? Section 2.4 complements this view by practical considerations: How is the Flashix file system be integrated into existing infrastructure? What kind of performance can one expect in comparison to other systems? This work is briefly put in perspective to related work in Section 2.5, a more detailed comparison follows in the individual chapters.

2.2 From Paths to Bytes

This section describes the boundary interfaces and the three implementation layers of Figure 2.1, briefly introducing all the concepts, data structures and algorithms necessary to understand how a modern flash file system works internally. The concepts are illustrated in several ways alongside the detailed system architecture and refinement hierarchy shown in Figure 2.2.

Following the refinement approach, each layer defines its own particular view and abstraction of the state that exists at runtime. The respective view is chosen to support reasoning about the concepts studied at that level. As shown in Figure 2.2, the hierarchy is *deep* as there are many different concepts involved, so that the data representation is switched often in the formal development.

Each refinement step encodes design decisions, which makes a part of the file system's state concrete and which keeps the remaining part abstract. Key to understanding how this works is that there is no state that is truly persistent in some sense, all flash access ultimate goes through the driver interface represented by an abstract formal model. The implication is that each concrete layer can only employ internal data structures that are stored in volatile main memory, and therefore power cut safety must be considered on all layers.

A similar description of modern flash file system concepts can be found in the UBIFS

white paper [88]. The interested reader is referred to this document for more details that are directly connected to the implementation and some additional aspects not covered here (such as disk quota and transparent data compression).

2.2.1 POSIX—The Specification

The role of the POSIX specification in Figure 2.1 is to define syntactic interface, and more importantly, the specification of the desired behavior of the Flashix file system. The associated formal model reflects precisely the textual standard [3] without conceptual simplifications in the interface so that it can be used by existing applications in practice. At the same time, the formal model has been designed to be highly abstract: the formalization (excluding a characterization of possible error codes) can be presented compactly on one sheet of paper (see Figure 7.2). This makes it easy to understand the correctness guarantees.

A POSIX compliant file system is a name space of directories and files that is addressed by paths. However, the POSIX specification has several features that go beyond a model of a file system as a simple tree.

It is permitted to reference a file under different paths by *hard-links*. This feature is useful for example to create backups without duplicating the file's content. Hard-links, however, lead to sharing in the file system and its structure becomes a directed acyclic graph (only the directory part remains a proper tree). File content is accessed indirectly through *file handles*. There may be several handles pointing to the same file at a time. Each file handle stores a current read/write offset into the file. As a consequence, there are two different kinds of references to a file, from the tree and from file handles, which must be tracked so that the resources can be released appropriately.

An important part of the POSIX standard is concerned with *error handling*. The textual standard specifies to a great detail, which error codes must or may be returned in which situations and it is necessary to reflect these in the formal model as well, see Section 7.3. Errors can be partitioned into two classes: Invalid user input such as the request to remove a file that does not exist, and internal errors that cannot be foreseen from the outside such as resource exhaustion or hardware failures. Flashix enforces the strong guarantee that in any case an unsuccessful operation must not modify the state in any observable manner (the textual POSIX standard [3] does not specifically require this and is often imprecise in this regard, although some explicit but weaker guarantees are given for e.g. `rename`).

Lastly, the formal POSIX model requires that the implementation is robust against *power cuts* by specifying that all operations must take effect atomically. However, it is not the case that such a crash is completely invisible from the outside even on this level of abstraction: any file handle that had been open at the time the power cut must be discarded because the proces owning the file handle ceases to exist. This potentially causes some cleanup of resources related to *orphaned files*, as described in Section 7.6.

2.2.2 VFS + AFS: Generic Concepts

A Virtual File System (VFS) is an abstraction layer that sits between the external interface exposed to application programs and a concrete file system implementation. It provides functionality that can be realized generically and independently of the data representation and internals of a concrete file system. This functionality covers for example name resolution/path traversal in conjunction with checking of access rights, handles to open

files, and the segmentation of file content conformant to virtual memory pages.[1] The VFS is a central point where optimizations like caching can be implemented uniformly.

Such an abstraction layer appeared for example in Solaris 2.0 [105] in 1992 with the goal of unifying local and remote access to files. Similar concepts have since been integrated into each major operating system: the corresponding component is called Virtual Filesystem Switch in Linux—with dozens of concrete implementations—and similarly Installable File System (IFS) in Windows.

The VFS relies on a concrete file system implementation for storing data on the device. To this purpose, it defines an internal interface, data model, and an associated contract through which it communicates with the concrete file system implementation. This interface decouples the internal data representation of the concrete file system from the externally visible path-based interface. In the following we refer to this interface and later and its formal specification as the Abstract File System (AFS). Any compliant concrete implementation can be plugged in, as long as it satisfies the contract of the AFS interface. In particular the flash file system core described in Chapter 9 is a verified implementation of AFS.

The formal development process follows the same decomposition: The proof that VFS correctly realizes POSIX concepts relies on the abstract specification in terms of AFS only but is independent of the flash specific parts, and conversely, further development does not need to reconsider the concepts addressed in VFS—several hard problems are solved once and forall, in particular, the mapping of the algebraic tree structure to a pointer-based representation (with all the well-known difficulties [24]), and a mapping from a linear view of file content as a sequence of bytes (which involves nonlinear arithmetic and several corner cases).

2.2.3 Flash Specific Concepts

The flash file system core in Figure 2.2 tackles the problem that updates need to be written *out-of-place* to flash to accomodate the limitations of the hardware (cf. Section 1.3). It is modeled after UBIFS [88], which is a *log-structured* file system [135]. All file system data is maintained uniformly in an unordered collection on flash memory. New data is always written to fresh locations and an index in main memory tracks the current version of a given datum. The index, however, is just a caching mechanism. It can be reconstructed from an outdated version that had been persisted earlier on flash, in conjunction with the information of recently written data that is kept in a log. The log must determine a sequential ordering of these recent writes to maintain the inter-dependencies between user-level operations. A direct solution for flash file systems is to write entries constituting the log linearly to a dedicated part of the storage medium, called the journal. Once this part becomes full it is integrated into the regular storage area, updating the flash index as well in an operation called commit. A new empty journal area is then allocated and the log is cleared.

The FFS core relies on two submodules: an efficient implementation of the index as a B^+ tree (Section 2.2.5) and the mapping of data to the block structure of the hardware, called the journal layer (Section 2.2.4). These are factored out and represented by their respective abstract specifications (just like the decomposition at the level of VFS).

The FFS core takes a central role in the assignment of responsibilities in the overall file

[1] These are distinct from the physical pages of the hardware.

system: it balances the narrow design space between the relatively strong requirements to satisfy the AFS specification it implements and the weak guarantees that the submodules can provide due to power cuts and general unreliability of the flash hardware. In the formal refinement, this manifests in the exact way how invariants and preconditions formulated and the order in which steps of an operation are performed.

The file system core implements safety against unexpected power cuts by two means with the help of its two submodules:

1. Data from all top-level operations is grouped into transactions that are encoded in a way that makes it possible to detect partially written groups in the log. Partial groups are discarded upon recovery.
2. Since the index is just a cache an invariant is maintained that roughly reads as $\mathit{ram\ index} = \texttt{replay}(\mathit{flash\ index}, \mathit{log})$.

Since replaying the log dominates startup time, the on flash index is re-written periodically during commit, incorporating all changes made to the ram index so far in an incremental way. The previous log simply becomes part of the ordinary collection of files and directories.

Due to out-of-place updates, data accumulates over time on the medium. It is necessary to clean up unreferenced version of data, comparably to garbage collection in (functional) programming languages. Some aspects can already be studied on this level of abstraction, for example, that only obsolete data is freed.

2.2.4 Transactional Journal

The journal layer of the file system writes objects representing file system data of the core as part of transactions that appear to take effect atomically with respect to power cuts. In order to guarantee this atomicity, the journal groups such objects per operation. Such groups must have been written in their entirety in order to make a valid contribution to the observable file system state. Atomicity at the level of individual writes of objects is required as well, but for the sake of modularization this concept is not addressed in the journal but in the persistence layer (explained below in Section 2.2.6), which handles serialization/deserialization and thus provides the transition down to a byte-based view.

The algorithms realized in the journal refer to the block structure of the hardware, partitioning the device into Logical Erase Blocks (LEB) that are mapped to physical counterparts by the Erase Block Management layer (described in Section 2.2.7). The content of each block is regarded as a simple sequences of data objects (which is the view provided by the persistence layer). This approach permits one to reason about transactions at a relatively high degree of abstraction without taking specifics of the on-disk layout into account.

At the same time, it becomes possible to reason about several aspects of free space management and allocation of erase blocks. In close cooperation with the index module, the journal layer keeps track of how many bytes in each block still contain live data. If this measure falls under a certain threshold or if storage space becomes scarce, a garbage collection procedure moves live data out of almost-empty blocks so that these can be erased and reused.

2.2.5 B$^+$ Tree Index

The index module realizes the indirection necessary to transparently move objects around on flash memory (by writing newer versions and through garbage collection). As viewed from the file system core, it is a simple mapping from stable object identifiers (keys) to the current address of the object on flash memory. The index is realized as a B$^+$ tree [18].

The first concern of the index is to support rapid queries and updates of this mapping by caching these operations as part of an in-memory data structure. The second concern is to support persisting this mapping to flash memory by the periodic commit. The size of the index grows linearly with the size of the file system, including the amount of storage allocated for file content. Therefore, it works *incrementally*: the in-memory representation is populated lazily from the underlying on-flash version during access, and conversely, during a commit, only the changed portion is written to disk.

B$^+$ trees lend themselves naturally to this mode of operation, since they are well-suited for large amounts of data. The on-flash index can be updated similarly to a functional data structure—this scheme is called "wandering trees" in the context of file systems [88].

2.2.6 Persistence Layer

The conceptual switch from high-level data structures towards a low-level byte-based view is centralized within the persistence layer. The two components for the journal and the index rely on this interface to store objects of the file system core respectively the nodes of the on-flash copy of the index.

More generally, the persistence layer defines the overall layout of the flash device, which is partitioned into several areas (the on-flash copy of the index, the main area storing the data from the journal, the log as a list of recently written erase blocks, and the orphan area). The current location of all of these areas is indexed by a superblock that is stored at a known location.

The persistence layer takes responsibility to encode and decode data structures in the different areas and to provide information about the sizes of serialized data (the upper "encoding" component in Figure 2.2). In order to protect against partial writes from hardware errors or power cuts, the serialization format includes additional headers and trailers that surround the actual data on disk. A missing or damaged trailer, for instance, signifies a partial write.

The persistence relies on a cache, the *write-buffer*, that mitigates the restriction of the flash hardware supporting page-aligned writes only. With such a cache, multiple requests to write some objects can be coalesced, although there will be some data pending in a partial, cached page in memory that has not yet been flushed to disk. In order to provide atomicity guarantees within the upper layers, the persistence interface encompasses an operation to synchronously write out this cache to the hardware, filling the remaining (wasted) space with a padding marker. Such holes in the on-flash format are eliminated by garbage collection later on, which always packs objects tightly.

2.2.7 Erase Block Management and Hardware Model

All flash access described so far is interpreted in terms of Logical Erase Blocks (LEB) that are mapped transparently to their physical counterparts in an on-demand fashion. The Erase Block Management (EBM) layer ("logical blocks" in Figure 2.2) provides an interface that is similar to the low-level operations of the hardware (read, write, and erase), except

that blocks are addressed logically. With an extra indirection between logical and physical blocks it is possible to implement several advanced features:

Logical blocks are allocated on-demand and mapped to physical ones. Erasing of physical blocks can be done asynchronously in the background, reducing the latency of the external operation to deallocate logical blocks (removing the mapping), which then appears to take effect immediately.

The EBM layer can change of a whole logical block apparently atomically in a way that is safe against power-cuts, simply by preparing a fresh physical one with the new content and then updating the mapping. This is employed for example to write a new superblock during a commit, when the different areas move around the storage space.

The EBM implements a technique called *wear-leveling* to prolong the lifetime of the device: As each physical block can endure only 10^4–10^6 erase cycles, these should be distributed evenly. This requires a proactive strategy. For instance, an almost full file system with a lot of stale data (that doesn't change any more) causes the other remaining available physical blocks to be repeatedly written and erased. To prevent this, the internal wear-leveling operation transparently migrates the stale data to a block that had been used heavily to give it some rest. As consequence, blocks with low erase counters that hadn't been used much in the past will be scheduled for future operations. Overall, running the wear-leveling algorithm causes the maximum distance between two erase counters to shrink.

The erase block management layer includes a submodule to encapsulate serialization and deserialization of organizational data that is internal to the EBM (lower "encoding" in Figure 2.2), notably an inverse mapping stored in the physical blocks to an assigned logical block number for allocated blocks.

As outlined previously, an abstract model of the flash hardware (resp. its driver interface, "driver spec." in Figure 2.2) contrasts the top-level POSIX specification as the bottom boundary of the overall development. The model encodes the assumptions stated in Section 2.3.1 in a concise manner and thus captures the semantics of the hardware operations to read, write, and erase pages and blocks subject to a no-overwrite policy.

2.3 The Verification Perspective

Purpose of this section is to answer the question: What does it mean that the correctness of Flashix has been formally verified? The answer to this question relates to the formal specification of the boundary interfaces, namely POSIX as the provided interface and MTD as the driver interface relied on. Here, the intentions are described that underlie the design of the formal models.

2.3.1 Assumptions

The assumptions about the hardware strike a balance between guarantees that the file system can rely on and the limited reliability actual hardware can provide.

A write or erase operation may take partial effect. This mirrors the fact that Flash hardware can sporadically show errors due to the somewhat unstable physical characteristics. It is assumed, however, that such as situation can be detected reliably from an error code returned by the driver interface. Furthermore, only the portion of the data accessed at the time of a write operation may be affected by an error, whereas previously written data is assumed to be safe.

For read operations, Flashix assumes that the state of the device is not changed at all. This assumption is not entirely realistic: Investigation of modern flash hardware has revealed the so called "unstable-bit" issue[2] or "read disturb" in [153, Section 4.1.3], where locations on the device that had been written at the time of a power cut may degrade with a small number of read operations. UBIFS tries to protect against such a situation by migrating away data from the affected blocks. Such a procedure has not been integrated into Flashix yet.

The final assumption is that the mechanism for bad block management is reliable. The assumptions stated above are encoded as part of the formal model of the driver interface described briefly in Section 2.2.7 and formally with more detail in Chapter 10.

Of course, the standard restrictions for verified software apply: a bug in any of the tools involved can weaken the result of the formal proofs. It is therefore assumed that the theorem prover (KIV [53] in this case) does not permit to derive invalid conclusions due to a problem in its inference engine, dependency management, and so on. Furthermore, the executable code is automatically derived from the formal models (see Section 2.4.1). Bugs in the code generator are not covered by the verification so far, this issue may addressed in future work. However, experience tells that such errors surface quickly and loudly in the form of a compilation failure or a runtime crash. They can be fixed easily most of the time as they tend to be systematic. The approach circumvents common programmer slips, too (such inadvertedly using the wrong comparison operator), because these are already caught at a higher-level in the formal models by the proofs.

2.3.2 Guarantees

Formal verification of the Flashix file system shows several desired properties of the implementation model, presuming that the formal POSIX model actually captures the standard [3] adequately.

Functional correctness means that the implementation behaves exactly as required by the top-level formal specification, it is established by refinement proofs. As a consequence, all internal data structures and algorithms supporting the flash file system concepts are shown to be realized correctly.

Proper handling of invalid input is enforced by the elaborate specification within the POSIX model.

The internal operations, namely garbage collection, commit, asynchronous erase, and wear-leveling, work as advertised and have no externally visible effect, even in the presence of hardware errors and take effect atomically with respect to power cuts. A successful garbage collection cycle of a logical block actually makes this block available.

The file system can recover from a power cut at any time. This implies that the resulting state will be consistent in the sense that all data structures in memory and on disk are well-formed and all the invariants hold. More specifically, recovery produces a state that is observably equivalent to either the one before or after the operation running at the time of the power cut. This guarantees in effect that the top-level interface is atomic with respect to power cuts.

The file system deals gracefully with errors of the hardware. Operations are safeguarded against write errors similarly to power-cuts and therefore the visible effect of such an error is a non-operation. Furthermore, write operations are potentially retried

[2] http://www.linux-mtd.infradead.org/doc/ubifs.html#L_unstable_bits

several times internally.

However, the file system does not have builtin redundancy that can mitigate read errors and any data that is stored in a broken location on the device will remain inaccessible. Some integrity measures are implemented, e.g., several on-flash data structures are guarded by checksums in addition to error correction provided by some hardware, although this is currently not done for all data stored on flash. Such extensions are considered orthogonal to this work and could be added modularly in the future. Nevertheless, hardware errors typically manifest during write or erase operations and can therefore be detected before bad locations are populated with data. In such cases, the bad blocks are excluded from further use before any harm is done.

The functional guarantees and the error behavior manifest in the formal model of POSIX in Chapter 7. The guarantees related to power cuts are established by interpreting the top-level specification as an atomic one in the first place—the formal refinement theory underlying the verification ensures that the final implementation adheres to the specified atomicity (in addition to the usual mechanism for functional correctness).

2.4 The Practical Perspective

This section discusses how the Flashix file system works in practice.

2.4.1 Integration, Running Code, and Validation

From the formal models we generate executable code in two programming languages, Scala and C. Both can be integrated into the Linux directory hierarchy via the File System in Userspace (FUSE) library[3] that permits one to implement a file system as a conventional user-space program. The two code generation targets differ in their purpose:

Scala[4] is a modern, scalable, multi-paradigm programming language that compiles to the Java Virtual Machine (JVM). It supports object oriented concepts that are used to encode the ASM models as classes with the given operations and an internal state. Scala emphasizes immutable data structures and functional programming and provides a comprehensive library of data types such as lists, sets and maps, which makes it easy to translate algebraic definitions from the formal specifications. The generated Scala code closely resembles the formal KIV models. The Scala code is used for simulation, testing, prototyping, and validation of the formal POSIX model. By running on top of the JVM, it is simple to integrate Java libraries for example to visualize data structures. An interactive demonstration has been developed that shows the status of erase blocks and the index for instance.

Targeting C code makes it feasible to run Flashix on actual embedded devices. The C code is much more efficient in terms of run time and memory in particular and can be integrated more easily with the FUSE and MTD interfaces. The code generator is limited to the implementation level models of the hierarchy (the grey ones in Figure 2.2). The generated code is more verbose (≈13 kLoC of C in contrast to ≈7 kLoC of Scala), for example, it needs to be augmented by explicit memory management and an encoding of algebraic data types as C structures and tagged unions. The C code must bridge the conceptual gap between the value semantics of the algebraic types towards destructive modifications. The specifics of the code generator are not subject to this thesis, though.

[3] https://github.com/libfuse/libfuse
[4] http://scala-lang.org

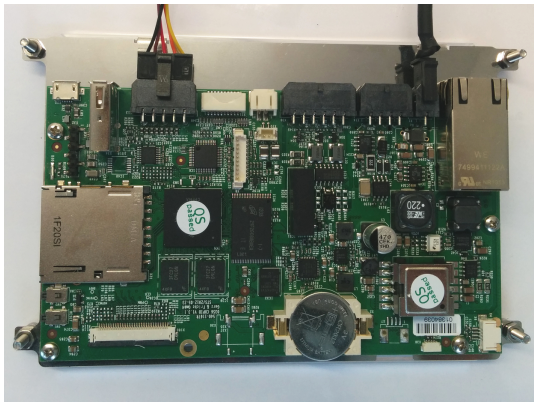


Figure 2.3: Garz & Fricke CUPID development board (14×8 cm^2).

The CUPID board shown in Figure 2.3 by Garz & Fricke is a small embedded system designed for experimental setups and prototype applications.[5] It is equipped with a Freescale ARM 11 processor with 532 MHz and 128 MB RAM and runs embedded Linux or Windows CE. An on-board chip provides 256 MB of NAND flash storage space consisting of 2048 erase blocks of size 128 KB with 64 pages of 2048 bytes each. The root file system is stored in one of several flash partitions and is formatted with UBIFS by default. The device is accessible via serial port and Ethernet. The CUPID board currently serves as test platform for the Flashix file system.

The guarantees that Flashix provides are necessarily relative to whether the formal POSIX model actually expresses the *intended* behavior of the system. Refinement proofs only guard against errors in this part of the development to a certain extent.[6] For this reason we have *validated* both the Scala code for the POSIX model as well as the C Code using the SibylFS framework by Ridge et al. [132] for oracle-based conformance testing. This activity has uncovered only minor issues with features beyond the scope of the Flashix project (notably missing support for symbolic links, which then was added manually to the generated C-code).

2.4.2 Performance

This section gives a glance at the performance of the C version of Flashix. Although a comprehensive evaluation of the file system under different workloads is beyond the scope of this thesis, some evidence is provided that Flashix is fast enough to be used in practice. Figure 2.4 shows a comparison to UBIFS+UBI for some microbenchmarks on a simulated flash device. The flash device provides 1 GB of storage and is simulated by the `nandsim`[7] kernel module that is distributed as part of Linux. The module is loaded with delays enabled. The benchmark encompasses an initial formatting of the device, a subsequent mount, writing a single larger file ($\approx$ 150 MB), writing many small files ($\approx$ 2500), and reading back each of the small files.

The measurements for UBIFS discern whether the `sync` mount option had been specified that forces all file content to be written to disk immediately. In the asynchronous

[5] http://www.garz-fricke.com/cupid-core_en.html

[6] It is possible that proofs for a correct implementation against a flawed spec uncover such problems. The issue described in Section 7.7 has been found this way.

[7] http://www.linux-mtd.infradead.org/faq/nand.html#L_nand_nandsim

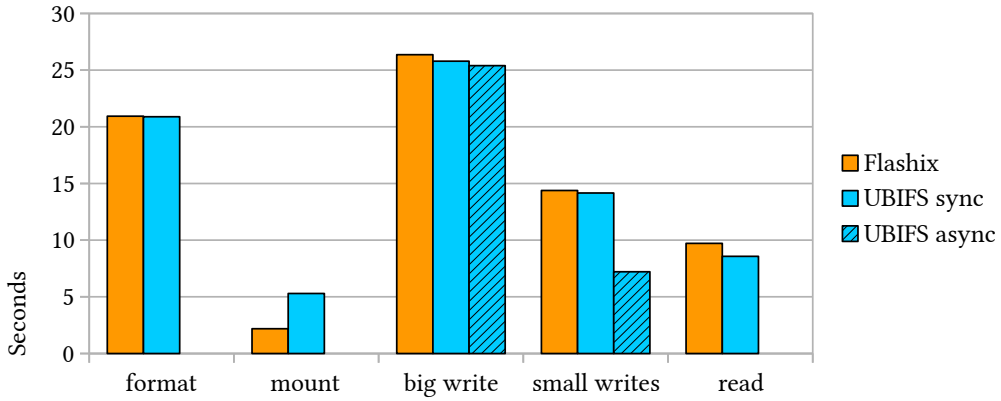


Figure 2.4: Comparison of Flashix and UBIFS for different microbenchmarks on a simulated device. The measurements have been taken with all optimizations of UBIFS and Linux disabled or bypassed (caches and compression in particular). The numbers are dominated by disk access times.

case without that flag, the measurements include a subsequent flush of internal buffers to disk (userspace command `sync`). The notable difference manifests for small writes that can be partially coalesced in the asynchronous case. The support for internal compression had been disabled for UBIFS, and the file system was unmounted and remounted between each test so that the elaborate caching in Linux does not influence the results.

The benchmark is not representative for a realistic workload and effectively measures just the I/O load generated, disregarding any optimizations in UBIFS/Linux not implemented by Flashix. The indicated numbers are dominated by the (simulated) access times of the flash hardware. The results shown in Figure 2.4 can be interpreted to demonstrate that the concepts, algorithms and data structures implemented in Flashix are not significantly less efficient than the ones in UBIFS, and that the refinement-based development method with generated code is a valid approach to produce systems that are both provably safe and without sacrificing much.

Under *realistic* workloads that necessarily take advantage of the more sophisticated internals of UBIFS and Linux (caching and internal concurrency), Flashix exhibits a performance that is slower by a factor of 2 to 5. Work to address this performance gap is already ongoing.

2.5 Summary of Related Work

This section concludes with a short survey of file systems for flash memory; and a brief discussion of related efforts in a formal context.

2.5.1 Existing Flash File Systems

The Journaling Flash File System (JFFS) for raw flash memory[8] targets devices of restricted size only. It trades efficiency in favor of a relatively simple design. For example, the index is reconstructed at mount time instead of being stored on flash, which makes initialization slow and prompts high memory usage because the RAM index must be total. JFFS 1 targets

[8] `http://sources.redhat.com/jffs2`

older NOR flash only, support for modern NAND has been built into version 2. JFFS is used for example in home routers, which often run embedded Linux. Yet Another Flash File System (YAFFS)[9] has less restrictions, but it relies heavily on out-of-band data storage, which is not always available on modern flash devices.

UBIFS improves significantly[10] on the performance of JFFS and YAFFS through advanced features: It supports for example incremental loading and storing of data structures such as the index and free space tables, transparent data compression, asynchronous internal operations, and write-back caching. UBIFS has been part of Linux since 2008 and is considered very mature. It is actively maintained and developed further at the time of writing, for example, it has recently grown support for Multi-level cell (MLC) NAND hardware, where memory cells have more than two states and can thus store multiple bits.

The flash file system LogFS is based on the idea to store the directory tree explicitly (similar to Ext2). It is updated out-of-place on flash, similar to a functional data structure or the wandering tree of the flash index of UBIFS. LogFS has been integrated into Linux in 2010, however, further development seems to have ceased by 2012.[11]

For devices with a Flash Translation Layer (FTL) there are several file systems that can issue TRIM commands to the device in order to mark space as unused. This helps the FTL to reclaim space that it could otherwise not determine as unused. TRIM is implemented in file systems originally designed for conventional magnetic disks, for example in Ext2 and its successors.

The Flash Friendly File System (F2FS)[12] is an experimental design by Samsung that takes the inner workings of FTLs into account and supposedly triggers less erase cycles. A similar file system is NILFS[13], which builds a snapshot mechanism on top of its log structured design.

2.5.2 Formal Approaches

An early pen-and-paper model of the "UNIX filing system" is by Morgan and Sufrin [108]. It is written in the specification language Z and considers a wide range of POSIX concepts such as hard links and file handles. This work has served as foundation and inspiration for a first generation of formal models that have been mechanized [10, 39, 40, 57, 61, 81, 92]. These approaches typically study some aspects on a high level of abstraction and in isolation. Although some efforts are made towards an implementation, none of these has produced a working prototype. Lali [98] summarizes these efforts.

A second generation of file systems research with formal methods aims at complete and realistic systems [8, 33, 34, 65, 93, 104, 120, 132, 133]. The distinguishing result of most of these efforts is running code that serves as a test oracle or provides an actual storage system. Furthermore, specification of the POSIX interface is revisited in the light of state-of-the art formal methods techniques, and safety against power cuts resp. system crashes is considered seriously. In this context, domain specific languages such as COGENT [9, 121] have shown the potential of bridging the gap between high-level models and low-level implementations.

A detailed and technical comparisons to all of these existing efforts is included at the

[9] http://www.yaffs.net
[10] http://www.linux-mtd.infradead.org/doc/ubifs.html#L_scalability
[11] https://github.com/prasad-joshi/logfs_upstream
[12] https://lwn.net/Articles/518718
[13] http://nilfs.sourceforge.net/en

end of the respective chapters as appropriate.

Orthogonal efforts apply model checking to existing real-world file systems with impressive results in terms of defects found [21, 63, 96, 112, 161].

A more general perspective includes large scale formal methods applications such as software of the Paris métro [19], Microsoft's hypervisor Hyper-V [36], the PikeOS microkernel [16], NICTA's microkernel seL4 [94, 95], the verified compilers CompCert [100] and CakeML [97], and the distributed system IronFleet [76] (this list is not intended to be exhaustive).

A comparative but smaller case study has been done for the Mondex electronic purse where several teams have provided different solutions to the verification of a secure protocol, see [90] for an overview. Another well-documented case study is the Tokeneer access control system [15].

Chapter 3

Background

Summary. The foundations of this work are structured specifications for algebraic definitions of data types such as numbers, lists and sets, and functions over these data types such as arithmetical operations, length, union, intersection and so on. The algebraic part is defined using higher order logic. The computational part of the models is given as transition systems over an algebraically defined state space, syntactically encoded as Abstract State Machines (ASMs). The interactive verification system KIV supports these formalisms and provides a weakest-precondition calculus to reason about properties of ASMs.

Contents

3.1 Algebraic Specifications

The foundations of this work are structured algebraic specifications and many-sorted higher-order logic [12, 53, 130]. It is based on typed expressions $e\colon t$, which composed of variables $x \in X_t$, typed function symbols $f\colon t_1 \times \cdots \times t_n \to t$, applications of a functional to several arguments $e(e_1, \cdots, e_n)$, and of lambda abstractions $\lambda\, x_1, \cdots, x_n.\ e$ Vectors of values/variables are typically signified by an underline $\underline{x} = x_1, \cdots, x_n$ as in $\lambda\, \underline{x}.\ e$. Function symbols that take no arguments ($n = 0$ in their type) are called constants. Expressions of the builtin type `Bool` denote formulas. Function symbols $p\colon t_1 \times \cdots \times t_n \to$ `Bool` are called predicates, these include `true`, `false`: `Bool` (writing these in `typewriter` font), the standard connectives $\neg, \wedge, \vee, \to, \leftrightarrow$ (in decreasing order of binding precedence), and the existential and universal quantifier $\exists\,, \forall : (t \to \texttt{Bool}) \to \texttt{Bool}$ as usual. Types are formed from a set of base sorts s and function types.

The semantics is based on algebras $\mathcal{A} = (A_s, \_^{\mathcal{A}})$ with a carrier set A_s for each base sort and a function $\_^{\mathcal{A}}$ assigning meaning to constants and function/predicate symbols f, written $f^{\mathcal{A}}$. The carrier set of function types $A_{t \to t'} = (A_t \to A_{t'})$ is defined recursively over the structure of the type as the set of all functions from A_t to $A_{t'}$ (standard semantics).

Expressions $e\colon t$ are evaluated with respect to an algebra $\mathcal{A}$ and a valuation $v\colon X_t \to A_t$

for the variables as follows:

$$\begin{aligned}
[\![x]\!](\mathcal{A}, v) &= v(x) \\
[\![f]\!](\mathcal{A}, v) &= f^{\mathcal{A}} \\
[\![e(e_1, \cdots, e_n)]\!](\mathcal{A}, v) &= a(a_1, \cdots, a_n) && \text{for } a = [\![e]\!](\mathcal{A}, v),\ a_i = [\![e_i]\!](\mathcal{A}, v) \\
[\![\lambda\ \underline{x}.\ e]\!](\mathcal{A}, v) &= \underline{a} \mapsto [\![e]\!](\mathcal{A}, v(\underline{x} \mapsto \underline{a}))
\end{aligned}$$

It is assumed that the connectives have their usual meaning, e.g., $\neg^{\mathcal{A}}$ is logical negation, and $\texttt{true}^{\mathcal{A}} = \mathit{true}$ resp. $\texttt{false}^{\mathcal{A}} = \mathit{false}$ for semantic values $A_{\mathtt{Bool}} = \mathbb{B} = \{\mathit{true}, \mathit{false}\}$ (written in *italics*). The semantics of a predicate $p\colon t_1 \times \cdots \times t_n \to \mathtt{Bool}$ can be regarded as a relation on the carrier sets $[\![p]\!] \subseteq A_{t_1} \times \cdots \times A_{t_n}$ (which is isomorphic to explicitly mentioning the target type's carrier set $A_{\mathtt{Bool}}$). This is used to shorten the notation in the later chapters.

A *specification* is defined in terms of a signature, fixing the sorts and function symbols that are available, and a set of axioms that characterize the properties of the functions. An algebra is a model of a specifications, if it satisfies all axioms for any valuation. The development in this thesis is to be understood as relative to such a model, which will be left implicit.

Specifications will be introduced step by step later on as part of the mathematical formalization of the data structures as well as the algebraic functions and predicates in the case study. Here, some of the commonly used data types and functions on them are defined in terms of specification fragments.

Frequent use is made of freely generated (algebraic) data types, introduced by the keyword **`data`**. For example, lists `List` are constructed from the constant empty list $[\,]$ and addition of an element at the front ("cons"), written with an infix `+`

$$\mathbf{data}\ \mathtt{List} = [\,] \mid \_ \mathtt{+} \_\ (\mathtt{head}\colon \mathtt{Elem},\ \mathtt{tail}\colon \mathtt{List}),$$

where the selectors $l.\mathtt{head}$ and $l.\mathtt{tail}$ retrieve the constructor arguments of a list l (written postfix by convention). $l.\mathtt{head}$ is the first element of a list l. `Elem` stands for any type of elements that is stored in the list, instances $\mathtt{List}\langle t\rangle$ where `Elem` is replaced by a type t are used as well. The length of a list is written as $\#l$, concatenation is written _ ++ _ and concrete lists are delimited by square brackets, e.g., $[a, b, c]$. The semantics of **`data`** types are given by the algebra that is freely generated over terms only containing constructors.

The type of streams $\sigma : \mathtt{Stream}\langle t\rangle$ generalizes lists to potentially infinite length. It is specified as a variant that is either a lists or a total function from natural numbers to element of the element type t

$$\mathbf{type}\ \mathtt{Stream}\langle t\rangle = \mathtt{List}\langle t\rangle + (\mathtt{Nat} \to t)$$

with a function $\#\sigma : \mathtt{Nat} + \{\infty\}$ to retrieve the length of a stream σ, prefix and postfix selectors $(\sigma\ \mathtt{to}\ n)$ resp. $(\sigma\ \mathtt{from}\ n)$ (defined for $n \leq \#\sigma$), and concatenation $\sigma_1 \mathtt{++} \sigma_2$.

Finite sets and maps (partial finite functions) are used frequently, too, written $s\colon \mathtt{Set}\langle t\rangle$ resp. $f\colon t \twoheadrightarrow t'$ with domain t and range t'. The empty set and map is uniformly written as $\varnothing$, cardinality of a set is denoted as $|s|$, and singleton sets and maps are written as $\{a\}$ resp. $[a \mapsto b]$. Map lookup (partial function application) and update is written with square brackets $f[a]$, and $f[a \mapsto b]$ respectively.

The disjoint union of two maps $f_1 \uplus f_2$ is defined only when the domains of f_1 and f_2 are disjoint (in the remainder all occurence of $\uplus$ are assumed to be defined).

$$(f_1 \uplus f_2)[a] = \begin{cases} f_1[a] & a \in \mathtt{dom}(f_1) \\ f_2[a] & a \in \mathtt{dom}(f_2) \end{cases}$$

Finite multisets $m\colon \mathit{Multiset}\langle t\rangle$ are used as well, where $\uplus$ denotes multiset sum. The characteristic function $\mathtt{count}(a, m)\colon \mathit{Nat}$ denotes the number of occurrences of a in m and the membership test $a \in m$ abbreviates $\mathtt{count}(a, m) > 0$. When convenient, multisets are treated as ordinary sets, in such cases an occurrence of m is taken to abbreviate the set $\{a \mid a \in m\}$. Multiset displays use delimiters $\wr \cdots \int$ that resemble the shape of a "bag" (which is an alternative name for the mutliset type).

A data type that occurs frequently in implementations is the type of arrays $ar\colon \mathit{Array}\langle t\rangle$ that is similar to partial functions, except that it has a length, written $\#ar$, instead of a domain. Lookup $ar[n]$ and modification $ar[n \mapsto b]$ is written again with square brackets and is defined for $n < \#ar$ (indices n are 0-based).

Finally, sometimes predicative types are used, such as arrays of a specific length, written $\mathit{Array}_n\langle t\rangle = \{ar\colon \mathit{Array}\langle t\rangle \mid \#ar = n\}$. In the actual development in KIV these are encoded by additional assertions and invariants.

3.2 Abstract State Machines

Abstract State Machines (ASM) [29, 72] are a framework for the description and development of state-based software systems. The (sequential) ASM thesis formulated by Gurevich [73] states that every (sequential) algorithm can be simulated by an ASM with the appropriate degree of abstraction: states correspond to mathematical structures and ASM steps correspond to the steps that are attributed informally but intuitively to the algorithm at the respective degree of abstraction. For this reasons the ASM formalism has been advocated by Börger and Stärk [29] and others as a tool for formal methods that is on one hand very flexible because it scales with the development process and on the other hand mathematically rigorous due to precise but simple foundations.

The steps of an ASM are defined by abstract programs that manipulate a vector of program variables storing instances of algebraic data types as the state of the machine. Programming and modelling concepts supported by ASMs include assignments, conditionals, local variables, parallel composition, sequencing, nondeterministic choice, loops, procedure calls and recursion as first class syntax known from structured programming languages (see e.g. [28]). Some extensions deal with distributed and concurrent machines, but these are not used in this work.

The ASM method is equipped with a notion of refinement [27, 138] for incremental development of systems starting from requirements down to the final code. There are several tools related to the ASM formalism for simulation and testing (such as CoreASM [56] and Asmeta [66]) and for formal verification (such as KIV [53], which is used in this thesis, see Section 3.6).

Definition 3.1 (Abstract State Machine)**.** An ASM $\mathtt{M} = (\underline{x}\colon \mathit{St}, \mathtt{Init}, p)$ has a predicate $\mathtt{Init}\colon \mathit{St} \to \mathit{Bool}$ to characterize initial states and a main program p that describes computational steps, which manipulate a vector of program variables $\underline{x}\colon \mathit{St}$ constituting the state space (also called "dynamic/controlled functions"). The computation induced by an

ASM starts in an initial state that satisfies `Init`. The program p is then repeatedly executed and the states encountered are recorded in a trace, called a *run*.

In the following, a specific class of ASMs is used with most of the features outlined above but with a restriction to parallelism at the level of assignments.[1]

ASMs may also refer to named procedures $\rho(\underline{x}; \underline{y})\{p\}$ with formal input parameters $\underline{x}$ and reference parameters $\underline{y}$ and body p. The declaration of such procedures is simply assumed to be given as part of the specification that sets up the background theory.

Definition 3.2 (Program syntax)**.** The syntax of programs p, q is defined by the following grammar.

program	p, q	$::=$	$\underline{x} := \underline{e}$	parallel assignment
		$\mid$	$p; q$	sequential composition
		$\mid$	**if** φ **then** p **else** q	conditional
		$\mid$	**choose** $\underline{x}$ **with** φ **in** p **ifnone** q	nondeterministic choice
		$\mid$	**while** φ **do** p	iteration
		$\mid$	$\rho(\underline{e}; \underline{z})$	procedure call

Assignments $\underline{x} := \underline{e}$ evaluate the right hand side expressions in parallel. An assignment to a (partial) function $f := f[a \mapsto b]$ is abbreviated as $f[a] := b$ (similarly for other indexed data types, notably arrays). The empty assignment is abbreviated as **skip**, which has no effect. Sequential composition $p; q$ executes the two subprograms in order. Conditionals **if** branch on a condition φ and execute either p or the (optional) alternative q depending on the outcome of the test. The **choose** construct is a nondeterministic version of a **let** that picks some values for new local variables $\underline{x}$ that satisfy condition φ and executes p (the condition is sometimes omitted when it is true). If there are no such values then q is executed instead. **while**-loops execute their body as long as a condition is satisfied. The statement **abort** abbreviates the nonterminating computation **while** `true` **do skip**. Finally, calls to named procedures ρ with actual input parameters $\underline{e}$ and reference parameters $\underline{z}$ are included. Standard local variable declarations can be encoded as

$$\textbf{let } \underline{x} = \underline{e} \textbf{ in } p \;\equiv\; \textbf{choose } \underline{y} \textbf{ with } \underline{y} = \underline{e} \textbf{ in } p^{\underline{y}}_{\underline{x}} \textbf{ ifnone skip},$$

where $\underline{z}$ are fresh variables and $p^{\underline{y}}_{\underline{x}}$ denotes the renaming of $\underline{x}$ to $\underline{y}$ in p to avoid conflicts when $\underline{x}$ occurs in $\underline{e}$. Note that **ifnone skip** is never executed here and we drop **ifnone** clauses in the following when they are irrelevant. Choice between two programs can be encoded as follows, where b: *Bool*, omitting the condition $\varphi \equiv$ `true` and the **ifnone** part, and programs are grouped by curly braces:

$$p \textbf{ or } q \;\equiv\; \textbf{choose } b \textbf{ in } \{\, \textbf{if } b \textbf{ then } p \textbf{ else } q \,\}$$

Well-formed programs satisfy the usual consistency conditions. Identifiers used in assignments and actual parameter lists must be in scope, i.e., they are either state variables of the ASM or they are bound by a **choose** or by the formal parameters of a procedure body. Typing constraints must hold, e.g., both sides of an assignment have the same type.

[1] Formal deduction in the presence of arbitrary parallelism is complicated, because two programs running in parallel can issue conflicting updates that have to be dealt with (cf. the proof system given by Stärk and Nanchen [149] and the clash-freedom check by Schellhorn et al. [146]).

3.3 Sequent Calculus

The proof system underlying the work in this thesis based on sequents $\Gamma \vdash \Delta$ as judgements (see Gentzen [67, 68]). The conjunctive antecedent Γ of a sequent is a list of formulas representing the assumptions which must imply one of several potential conclusions in the disjunctive succedent Δ. A sequent $\Gamma \vdash \Delta$ is thereby an abbreviation of the formula $\bigwedge \Gamma \to \bigvee \Delta$. The proof system is syntax directed, there is exactly one proof rule for each propositional connective for the antecedent resp. succedent. As an example, the rules for conjunction are

$$\frac{\varphi,\ \psi,\ \Gamma \vdash \Delta}{\varphi \wedge \psi,\ \Gamma \vdash \Delta}\ \wedge\text{-left} \qquad \frac{\Gamma \vdash \varphi,\ \Delta \qquad \Gamma \vdash \psi,\ \Delta}{\Gamma \vdash \varphi \wedge \psi,\ \Delta}\ \wedge\text{-right}$$

The intuition is that assuming a conjunction amounts to assuming both its constituents, whereas proving a conjunction amounts to proving both constituents individually in two sub-proofs. The rules for disjunction are symmetric—the case split occurs in the antecedent.

The logic provides three modalities to reason about programs: Formula $\langle\!|p|\!\rangle\varphi$ denotes that program p is guaranteed to terminate when started in the current state and φ holds in all final states. This operator corresponds to the weakest precondition [44] of p wrt. φ and encodes total correctness. Formula $[p]\varphi$ denotes that whenever p terminates, φ holds in the final state. This operator is the weakest liberal precondition and encodes partial correctness. The sequent $\varphi \vdash [p]\psi$ corresponds to the Hoare triple $\{\varphi\}\ p\ \{\psi\}$. Dually, formula $\langle p\rangle\varphi \equiv \neg[p]\neg\varphi$ asserts the existence of some execution of p that terminates and establishes φ. For deterministic programs, $\langle\!|\_|\!\rangle\_$ and $\langle\_\rangle\_$ equivalent. The concrete notation is borrowed from [74, 127]. The advantage of these operators over Hoare triples is that several executions of different programs can easily be related, as exploited by the proof obligations for refinement in Section 5.2.

The calculus symbolically executes programs in modalities. Similar to propositional connectives, there are rules for each program construct. As an example, assignments introduce an equation

$$\frac{\Gamma, \underline{x}' = \underline{e} \vdash \varphi^{\underline{x}'}_{\underline{x}}, \Delta}{\Gamma \vdash \langle\!|\underline{x} := \underline{e}|\!\rangle\varphi, \Delta}\ \text{where } \underline{x}' \text{ fresh} \tag{3.1}$$

where fresh variables $\underline{x}'$ that capture the new values and for the postcondition $\varphi^{\underline{x}'}_{\underline{x}}$ denotes the corresponding syntactic renaming of $\underline{x}$ to $\underline{x}'$ in φ. The calculus is described in e.g. [79, 130].

3.4 Refinement of State-Based Systems

Refinement [157] is the idea of gradually developing a system in several stages or prototypes. One starts with an abstract possibly not feature complete description of a system that is elaborated and broken down towards the final implementation. There are different understandings of what a refinement step may comprise, for example the engineer may introduce new functionality (sometimes called "horizontal refinement") or replace an abstractly specified algorithm or mathematical data structure by their implementation counterpart ("vertical refinement").

State-based systems maintain an internal state that is not supposed to be accessible directly from the outside. Instead, an (external) interface is provided to mediate access to

the stored information, for example, to prevent unintended or malicious modifications that threaten internal consistency. The manipulation of the state is therefore fully under the control of the system.

While the refinement method was originally intended as a development guideline, it has quickly been realized that a formal underpinning of refinement leads to a useful strategy in program verification [44, 83], complementing for example the assertional methods by Floyd [59] and Hoare [82]. Specifically, when an abstract (description of a) system A is refined to a more concrete one C, one can take A as the specification of C and formally prove that the more complex C works the same as the easy-to-understand model A. In the following, we write $A \sqsubseteq C$ when an abstract system A is (formally) refined by a concrete one C, resp. when A specifies C, i.e., when A specifies the requirements for C.

With a suitable definition of the refinement relation $\sqsubseteq$, one can use A to reason abstractly about the behavior of the system: Although the actual running code will be provided by C, the conclusions from that reasoning still hold. To this end, refinement compares the *externally* visible behavior that can be observed over the interface but hides the internal state. This ensures that A and C can be regarded as essentially equivalent (where "essentially" is to be made precise).

A minimal requirement on the refinement relation $\sqsubseteq$ is that it is reflexive so that $A \sqsubseteq A$, i.e., A may serve as its own specification. Refinement should further be transitive so that $A \sqsubseteq B$ and $B \sqsubseteq C$ implies $A \sqsubseteq C$, i.e., one can take the liberty to develop a system in as many refinement steps as desired.

The fundamental question is therefore what can be observed about a system from the outside. This question can be addressed from two slightly different angles, depending on what is taken as the basis for observations.

The direct approach characterize a class of clients or contexts which may interact with the systems. The possible observations are implicitly given by the interaction of the system with its context. This view underlies the early definition of data refinement, which [77] succinctly states as

$$A \sqsubseteq C \quad \Longleftrightarrow \quad \text{for all programs } P : P(A) \equiv P(C)$$

where systems A, C are called "data types" and contexts are sequential programs P calling operations that constitute the interface of the respective data types. The equivalence $\equiv$ between programs is meant to be a semantic one, i.e., the problem of comparing the two systems A and C is reduced to comparing the outcomes of executing $P(C)$ to those of executing $P(A)$. For nondeterministic programs this equivalence is weakened to the set inclusion $P(A) \supseteq P(C)$ on the pairs of initial/final states. The benefit of this approach is that it is immediately clear, what refinement implies at the level of contexts.

An alternative approach introduces observations explicitly. For example, trace refinement of action systems [14] captures the behavior of a system A as the possible sequences $beh(A)$ of actions that it can engage in. Refinement $A \sqsubseteq C$ is then succinctly stated as $beh(C) \subseteq beh(A)$ (note the similarity to data refinement—again, set inclusion is used to permit that C is more deterministic than A). The minor drawback of this approach is that a composition theorem that relates refinement to contexts must be proved explicitly, which is demonstrated in [14] for parallel composition that synchronizes the actions of A resp B with a context M: $A \sqsubseteq B$ implies $M \| A \sqsubseteq M \| C$. The proof is based on monotonicity of the parallel composition operator $\|$ with respect to behaviors *beh*.

A syntactic approach is taken by the refinement calculus [13] that is based on Dijkstra's weakest precondition operators [44], which takes as observations the total correctness contracts one can prove about a system (which is a program in this case), i.e.,

$$A \sqsubseteq C \iff \forall\, \varphi.\ \langle\!|A|\!\rangle \varphi \to \langle\!|C|\!\rangle \varphi.$$

This formulation preserves any (total) correctness assertion made about the system in a refinement step and also between programs $P(A)$ and $P(C)$ containing A resp C as fragments. Liskov's substitution principle [101] and Meyer's design by contract [107] for object oriented programming narrow this down to a specific, given pair of precondition and postcondition for each method.

Explicit observations have the benefit that it becomes clear what a system means when viewed in isolation: There is no need to look at contexts in order to understand how a system works or in order to prove that a refinement holds.

Refinement of Abstract State Machines (ASMs) [27, 138] is based on comparing the runs of a system. However, in contrast to extracting the behavior of a system (an ASM) into observations that are separated from the state, one can specify explicitly, which part of the state constitutes the relevant information. A relation IO the fixes the desired correspondence between the abstract state of A and the concrete state of C, which permits for instance to refine the domain of inputs/outputs, i.e., to switch from mathematical numbers to a machine representation (cf. [42] for a similar concept in the specification language Z). Furthermore, the generalization to $n{:}m$ refinement permits to choose the points in time when the system's state is to be compared, while other states are excluded from the comparison. The ASM refinement method is thus much more flexible in comparison to many other approaches. However, it is also less clear how refinement relates to different patterns of composing systems, and arguments for modularity are somewhat ad-hoc and informal (see [27] for examples and [52, 162] for discussions).

There are some further technical decisions involved when setting up a refinement theory that are related to potential nontermination or infinite behaviors, see [139] for a discussion. For background on data refinement and related methods see de Roever and Engelhardt [41], Derrick and Boiten [42].

In summary the central aspect of a refinement method is which observations are compared, and how they are related to contexts that interact with the system. A major goal is to ensure that a proof of $A \sqsubseteq C$ guarantees that C can be substituted for A in any suitable context without violating the expectations of such a context on the behavior of its subsystem. The conclusion of this section is therefore that substitutivity is therefore a prerequisite for a *modular* decomposition of the development.

3.5 Separation Logic

Separation logic [131] is a logic designed to reason about pointer structures and destructive updates. It is particularly well-suited for structures with limited or no aliasing such as linked lists and trees. The presentation below is to be taken as a syntactic one, i.e., an algebraic specification, resembling the embedding of separation logic into KIV [53].

Formulas in the logic are assertions $\varphi\colon \mathit{Heap} \to \mathit{Bool}$ about the shape of heaps, which are mappings from locations to values, $\mathit{Heap} = (\mathit{Loc} \rightharpoonup \mathit{Val})$. The distinguished value $\mathtt{null}\colon \mathit{Loc}$ denotes an invalid location and is never in the domain of any heap. Heap assertions are built from the constant $\mathtt{emp}$ denoting an empty heap, the points-to predicate

$l \mapsto v$ describing singleton heaps, and the separating conjunction $\varphi \star \psi$ that asserts that the heap can be split into two disjoint parts satisfying φ resp. ψ.

$$\begin{aligned} \mathtt{emp}(h) &\leftrightarrow h = \varnothing \\ (l \mapsto v)(h) &\leftrightarrow h = [l \mapsto v] \\ (\varphi \star \psi)(h) &\leftrightarrow \exists\, h_1, h_2.\ h = h_1 \uplus h_2 \wedge \varphi(h_1) \wedge \psi(h_2) \end{aligned}$$

Iterated separating conjunction over a finite set $X = \{x_1, \ldots, x_n\}$ for asserts a formula $\varphi(\_)$ for each choice of $x \in X$ on its own part of the heap.

$$\circledast_{x \in X} \varphi(x) \quad \leftrightarrow \quad \varphi(x_1) \star \cdots \star \varphi(x_n)$$

Abstraction of pointer structures to algebraic counterparts is straight forward. The classic example refers to singly linked lists, where the objects of type *Val* stored in the heap are instantiated by nodes of the form

data *Node* = node(value: *Elem*, next: *Ref*)

The predicate ls correlates a pointer l to a list *Node* with an algebraic *List* as follows

$$\begin{aligned} \mathtt{ls}(l, [\,]) &\leftrightarrow (l = \mathtt{null}) \wedge \mathtt{emp} \\ \mathtt{ls}(l, a + x) &\leftrightarrow \exists\, r.\ l \mapsto \mathtt{node}(a, r) \star \mathtt{ls}(r, x) \end{aligned}$$

The recursion is well-founded over the algebraic data type. This definition states that an empty algebraic list is represented by a null-pointer in an empty heap. A list $a + x$ with head a and tail x corresponds to at least one heap cell $l \mapsto \mathtt{node}(a, r)$ with that element and some next pointer r to the remaining elements. The separating conjunction states that the first cell is disjoint from the remaining list, which precludes cycles globally.

With separation logic it is thus straight forward to specify recursive heap-data structures without sharing (proper trees in the general case) and to abstract them to an algebraic counterpart. The benefit of this approach is that whenever we have an assertion $(l \mapsto v) \star \varphi$ for some heap h, it is already known where an assignment $h[l] := v'$ must take place, because l cannot possibly alias any location in the φ-part due to the disjointness induced by the separating conjunction $\star$. Essentially, a syntactic assignment rule can be recovered and the resulting heap assertion will be $(l \mapsto v') \star \varphi$ regardless of φ—without separation one would explicitly need to prove that φ is preserved. This idea is codified by a "small" context-insensitive axiom for heap assignments and the *frame rule*, which permits to place a program running in a subpart of the heap into a bigger context, given by a frame F.

$$\frac{}{\{l \mapsto v\}\ h[l] := v'\ \{l \mapsto v'\}}\ \text{store} \qquad \frac{\{P\}\ p\ \{Q\}}{\{P \star F\}\ p\ \{Q \star F\}}\ \text{frame}$$

The proofs of the case study rely on a straight-forward shallow encoding of separation logic as a set of higher-order operators that has been mechanized in KIV in the course of this work. An example involving binary trees and further details can be found in [53]. Similar approaches are described in [116, 154]. Later on, the theory is applied to the mapping of a pointer-based directory tree to its abstract counterpart of POSIX (Section 8.8).

Two technical issues had to be addressed to make the approach work out. The first is concerned with lifting connectives and quantifiers over the heap parameter. For example,

conjunction $\wedge_\mathrm{h}$ between heap assertions is characterized by $(\varphi \wedge_\mathrm{h} \psi)(h) \leftrightarrow \varphi(h) \wedge \psi(h)$. In practice, the $\_\mathrm{h}$ subscript is omitted in favor of overloading the symbol $\wedge$.

The second technical issue is that there is no built-in heap in KIV resp. its support for ASM programs. Instead, the heap h is maintained explicitly as an ordinary program variable. This decision is a trade-off: with explicit heap variables it is possible to relate different (versions and types of) heaps, which is useful for specifying and proving refinements (as observed in e.g. [109]). The weakest-precondition calculus already supported by KIV had to be adapted just a bit for efficient proof support by a modified assignment rule.

The drawback of an explicit heap as part of programs is that the frame rule for heap-modular reasoning is not generally valid. Separation logic has been crafted carefully that all actions are local, specifically that all heap locations accessed by a program must be listed as part of the precondition; whereas in our setting non-local assignment such as $h := \varnothing$ are not ruled out in advance. A systematic fix for this lack of implicit framing is to justify it for each progam p in an ad-hoc manner: instead of proving $\{P\}\ p\ \{Q\}$ and lifting it by the frame rule, we always show $\{P \star F\}\ p\ \{Q \star F\}$ directly for a placeholder frame F, which can be instantiated by an arbitrary context later on. Our experience is that this solution imposes no additional effort in practice when the actions of p indeed affect the heap only locally, which is almost always the case. This section concludes with the sequent calculus rule for heap assignments that is now part of KIV, expressed over program modalities:

$$\frac{((l \mapsto v') \star F)(h'),\ h[l] = v \vdash \varphi_h^{h'}}{((l \mapsto v) \star F)(h) \vdash \langle\!|h[l] := v'|\!\rangle \varphi} \quad \text{where } h' \text{ is fresh}$$

and equally for the other modalities $[\_]\_$ and $\langle\_\rangle\_$. In the premise, the fresh variable h' denotes the new heap to which the new assertion and updated postcondition $\varphi_h^{h'}$ refer to. The additional assumption $h[l] = v$ records the previous value v at l so that no information is dropped by the rule and no harm is inflicted by automatically applying this rule during symbolic execution.

3.6 The Verification System KIV

The mechanization and proof support is provided by KIV [53, 130], a tool for the development and formal analysis of software systems. It is actively developed at the University of Augsburg and has been employed in several large scale formal projects such as the Mondex Challenge [90] and is also applied to the development of Flashix. KIV is used for teaching formal methods to students, too. Teams using KIV have participated successfully at verification competitions with the goal of comparing state of the art technology in the field.[2] The system is available at the site `http://isse.de/kiv`.

In its latest version, the KIV system supports several formalisms, notably functional and applicative models, imperative and concurrent programs, Abstract State Machines; associated with an integrated proof strategy that is based on symbolic execution for sequential and concurrent systems [145]. As a result of the work for this thesis, KIV now has explicit modeling and proof support for the refinement theories developed here.

KIV provides some advanced features that are specifically geared towards incremental development of big systems starting from a formalization of requirements down to realistic

[2] See `http://vscomp.org` and `http://www.verifythis.org`.

programs. The proof system provides interactive discovery of proofs on one hand and powerful automation on the other hand, scaling from initial prototypes and exploration (e.g. of system invariants) to maintenance and extensions of existing systems. A dependency management guarantees consistency between specifications, theorems and their proofs across multiple projects that tracks and minimizes impacts of changes to a development, which proved crucial for this project.

A graphical user interface that facilitates and speeds up development. It is complemented by an integration of project management and the specification editor into the Eclipse software platform,[3] with modern features such as syntax highlighting and incremental parsing and error checking of source files.

[3] http://eclipse.org

Chapter 4

Hierarchical Components

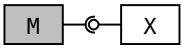
It's bigger on the inside! (comment on the TARDIS)

Summary. Components are represented by a specific class of Abstract State Machines (ASMs) that resemble data types: they have an interface given explicitly by operations with inputs, outputs and a precondition instead of just a main program. The theory is based on a fine grained view in terms of intervals that captures uniformly the semantics of components in terms of their runs as well as the steps of programming, building the foundations for the treatment of power cuts later on in this thesis and a compositional refinement method that supports abstraction of atomicity.

Publications: This chapter is based on [52].

Contents

Purpose of this chapter is to introduce a component model amenable to hierarchical composition, expressed in terms of ASMs that provide externally callable operations as their interface. Recall the diagrams for *submachine composition* from the introduction in Section 1.5 that describe the static structure of a system's development. In the following, the first diagram will be explained and formally defined, whereas refinement is postponed to Chapter 5.



submachine composition

Submachine composition is subject to the restriction that the two components may not depend directly on the other's internals. A machine M synchronously calling the operations of its submachine X is denoted by M(X) respectively by M—©—X graphically.

The view of ASMs in the remainder of this thesis is streamlined so that it matches more closely the intuition of interfaces defined by operations. Therefore, machines provide such an interface with top-level operations $\mathtt{Op}_j$ with an index j (a name) and explicit parameters, each defined by a program p_j. These machines will be referred to as "data type like" in the following. Graphically, one can interpret interface provision as the right half ○—X of submachine composition.

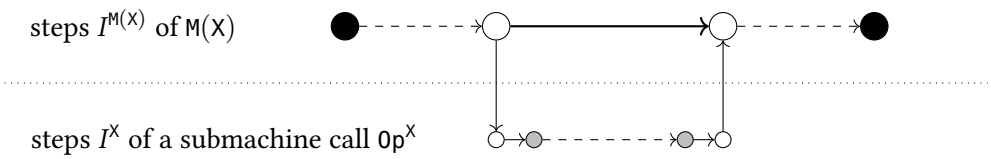


Figure 4.1: Dynamics of submachine calls. Several internal steps of $\mathtt{Op}^{\mathtt{X}}$ at the bottom are abstracted to an atomic call in the context of the caller (fat arrow at the top), hiding the grey states at the bottom.

Definition 4.1 (Data type like Abstract State Machine)**.** A data type like Abstract State Machine $\mathtt{M} = (\underline{x}\colon \mathit{St}, \mathtt{Init}, \{\mathtt{Op}_j\}_{j\in\mathcal{J}})$ has state variables $\underline{x}\colon \mathit{St}$, a predicate $\mathtt{Init}\colon \mathit{St} \to \mathit{Bool}$ to characterize initial states, and of operations $\mathtt{Op}_j$ for indices $j \in \mathcal{J}$. Each operation $\mathtt{Op}_j = (\mathtt{pre}_j, \mathit{in}_j, p_j, \mathit{out}_j)$ consists of an ASM program p_j that describes possible state transitions, provided precondition $\mathtt{pre}_j$ holds. It reads input from of formal input parameters in_j, and writes output formal output parameters out_j.

Data type like ASMs resemble standard ones from Definition 3.1, except that they have multiple operations instead of a single main program. It should be emphasized that this specific view is merely a shortcut to express certain facts and properties of composed systems in a direct way and to facilitate reasoning about compositionality and later on power cuts. It is, however, not an essential deviation from standard ASMs that have one main program only. This means in particular that the notation $\circ\!\!-\!\!\mathtt{X}$ denotes just the (data type like) machine $\mathtt{X}$, but it emphasizes that $\mathtt{X}$ is subject to use by some context. An embedding of such machines into the regular ASM formalism as described by Schellhorn [139] determines the semantics of data type like ASMs by transition systems (see Definition 4.11), although we'll capture the index of called operations and their inputs and outputs explicitly as part of their runs (cf. Section 3.4), whereas standard ASMs rely on input and output functions that are implicitly set by the environment.

The aspect that does in fact go beyond the standard ASM formalism is to admit *submachine calls* $\mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{z})$ with actual inputs $\underline{e}$ and outputs $\underline{z}$ to a submachine $\mathtt{X}$ in the context of a supermachine $\mathtt{M}(\mathtt{X})$. Dually to provision of an interface, this corresponds to its *use*, i.e., the left half $\mathtt{M}\!-\!\subset$ of submachine composition, also denoted by $\mathtt{M}(\_)$, which stands for a *partial* construction that is not a valid machine on its own until the missing submachine is linked in as described in Section 4.3.

In accordance with the interpretation of interfaces as establishing a *contract* between the caller and the callee [107], it must be ensured that the former respects information hiding and also the precondition of submachine: The submachine's state is accessed indirectly only via the inputs and outputs of its operations and the verification must check the respective preconditions at call sites. This renders submachines in an outer context as passive components like data types [77], in contrast to the prevalent view of ASMs as active components that internally choose their next steps (and thus have guards that determine applicable operations). This distinction is indeed required for builtin modularity as established by the modular refinement in Section 5.3.

On a technical level, the theory relies on a dual interpretation of sequences of states, called intervals I in the following: As the semantics of programs $I \models p$, expressing that I corresponds to the steps of a possible execution of p, and as the semantics of machines $I \in \mathit{runs}^{\mathtt{M}}(\underline{j}, \underline{i}, \underline{o})$, expressing that I corresponds to a run of $\mathtt{M}$ where the sequences $\underline{j}$, $\underline{i}$,

and $\underline{o}$ denote the called operations, their inputs, and outputs, respectively..

The main idea is to integrate submachine calls $\mathtt{Op}_j^{\mathtt{X}}(\underline{e};\underline{z})$ of a subcomponent $\mathtt{X}$ as atomic steps into the run of a caller $\mathtt{M}(\mathtt{X})$ as shown in Figure 4.1: The fine-grained steps of the program defining $\mathtt{Op}_j^{\mathtt{X}}$ are collapsed to a single atomic one in the context. The benefits of this approach are:

- The steps $I^{\mathtt{M}(\mathtt{X})} \models p$ of a supermachine program can be projected to a run $I^{\mathtt{X}}$ of the submachine $\mathtt{X}$ indicating which operations of $\mathtt{X}$ had been called. As a result, refinement expressed in terms of these runs gives rise to a corresponding substitution theorem in Chapter 5, which replaces $\mathtt{X}$ by a refined machine.
- From the small steps $I \models p$ of the execution of a program one can extract the intermediate states that are relevant for the analysis of power cuts (see Chapter 11).
- There is a well-defined place to change the degree of atomicity of steps at call sites by collapsing the micro steps of the underlying programs to a macro steps of the called operation (thus being bigger on the inside). This provides a lever for a systematic abstraction of power-cut safety.

The remainder of this chapter introduces the necessary concepts and definitions bottom up. The fine-grained nonatomic semantics of programs, the coarse grained atomic semantics of operations $\mathtt{Op}$, data type like ASMs (interface provision), and finally the integration of two machines $\mathtt{M}(\mathtt{X})$ with submachine calls (interface use).

4.1 Semantics of Programs

An interval $I \in St^\omega$ is a finite or infinite sequence of states in St. The length of an interval $\#I \in \mathbb{N} \uplus \{\infty\}$ reflects the number of transitions. If I is finite it consists of $\#I + 1$ states. In particular, the smallest intervals with $\#I = 0$ have one state only. The k-th state of an interval is denoted by $I(k)$, where $I.\mathit{first} := I(0)$ and $I.\mathit{last}$ is the last state of a finite I.

Sequential composition of finite intervals $I_1 \fatsemi I_2$ is defined when $I_1.\mathit{last} = I_2.\mathit{first}$ or when I_1 is infinite. It is defined by $I_1 \fatsemi I_2 := (I_1(0), \cdots, I_1.\mathit{last}, I_2(1), I_2(2), \cdots)$ for finite I_1 so that the common state is collapsed in the result. If I_1 is infinite, then $I_1 \fatsemi I_2 := I_1$ reflects the intuition that I_1 corresponds to a computation taking infinite time and I_2 will never be reached. The operator $\fatsemi$ is taken from Interval Temporal Logic (ITL) [31] where it is called "chop".

In addition to infinite intervals that model nonterminating programs, the integration of subcomponents requires to expose divergence of calls over an interface explicitly. Divergence is represented by states $\bot$, which can potentially occur at the end of intervals, but only there. This means that $I(k) = \bot$ implies that $I(k') = \bot$ for all $k \le k' < \#I$. Intervals I that are either infinite or end in $\bot$ both represent divergence, denoted by $I^\uparrow$ in the remainder. Keeping some additional $\bot$ states at the end of an interval provides some freedom of choice for its length. Care will be taken that the semantic definitions are robust against such an extension at the end.

In the following, by convention $(s_0, s_1, \cdots)$ with trailing dots refers to infinite intervals, correspondingly $(s_0, \cdots, s_n)$ with a specified final state s_n denotes a finite interval of length n.

"Ordinary" sequences $\underline{a} \in A^\omega$ over a set A are underlined by convention, called vectors when they are finite. Notation $|\underline{a}|$ counts the number of elements of $\underline{a}$ (contrasted to $\#I$ counting steps). Concatenation of sequences does not collapse the intermediate state and

is written by juxtaposition $\underline{a} = \underline{a}_1\underline{a}_2$. Sometimes, sequences of sequences resp. sequences of value vectors are used, these are written in boldface as $\underline{\boldsymbol{a}} = (\underline{a}_0, \underline{a}_1, \cdots)$

Modification $s(\underline{x} \mapsto \underline{a})$ of a state s at a vector of variables $\underline{x}$ to new values $\underline{a}$ is written with round parentheses like lookup. If $s = \bot$ then any modification will again give $\bot$. Similarly, modification of the whole trace of $\underline{x}$ in an interval I to a sequence of value vectors $\underline{\boldsymbol{a}}$ is written $I(\underline{x} \mapsto \underline{\boldsymbol{a}})$ such that $I(\underline{x} \mapsto \underline{\boldsymbol{a}})(k)(\underline{x}) = \underline{\boldsymbol{a}}(k)$.

Recall that the definition of the program syntax from Section 3.2

$$\begin{array}{rcl} p, q & ::= & \underline{x} := \underline{e} \mid \rho(\underline{e}; \underline{z}) \mid p; q \mid \texttt{if}\ \varphi\ \texttt{then}\ p\ \texttt{else}\ q \mid \texttt{while}\ \varphi\ \texttt{do}\ p \\ & \mid & \texttt{choose}\ x\ \texttt{with}\ \varphi\ \texttt{in}\ p\ \texttt{ifnone}\ q \end{array}$$

consists of assignments, procedure calls, sequential composition, conditionals, while loops and nondeterministic choice. The denotational semantics $I \models p$ as shown in Definition 4.2 records the steps of p within I that is finite whenever p terminates and infinite otherwise. The judgements $I \models p$ are derived by a (recursive) inference system as usual. However, in order to be able to derive infinite intervals for nonterminating programs, the commonly used least fixpoint is inadequate. Instead, the *greatest* fixpoint is used, which admits the construction of such intervals through infinite derivations. This approach is fairly standard [22, 23, 115].

The semantics of parallel assignments $\underline{x} := \underline{e}$ in rule (4.1) consists of one step from s to $s(xs \mapsto [\![\underline{e}]\!]_s)$ that overwrites the values of $\underline{x}$ in the new state with the denotation $[\![\underline{e}]\!]_s$ of $\underline{e}$ in the first state s, i.e., all expressions $\underline{e}$ are evaluated in parallel.

Sequential composition $p; q$ is defined in terms of concatenation of intervals. There are two cases, depending on whether p diverges (rules (4.2) and (4.3)). Recall that $I^{\uparrow}$ signifies that p does not terminate normally, i.e., $\#I = \infty$ or alternatively that the last state is $\bot$, which can arise from a diverging submachine call, see Section 4.3. Conditionals evaluate the test φ in the first state $I.\mathit{first}$ of the interval and then execute one of the branches.

The **choose** construct selects a sequence $\underline{\boldsymbol{a}} = (\underline{a}_0, \underline{a}_1, \cdots)$ of value-vectors $\underline{a}_k$ for the variables $\underline{x}$ so that $\underline{x} \mapsto \underline{a}_0$ in the first state establishes condition φ. The sequence $\underline{\boldsymbol{a}}$ in rule (4.6) fixes the values of $\underline{x}$ throughout the entire interval I during the execution of p and masks the previous value of $\underline{x}$ from the outer scope. If φ is unsatisfiable in the first state then the alternative q is executed instead by rule (4.7).

The **while** loop discerns whether φ holds in the first state of the interval. In the case φ holds, the loop is unfolded once, executing p and then the loop again if p terminates (rule (4.2)). Otherwise, the diverging body alone determines the interval (rule (4.8)). If φ is falsified, the loop is exited with a **skip** step (s, s) by rule (4.9). The definition effectively splits the interval $I = I_1 \fatsemi I_2 \fatsemi \cdots$ into segments $I_k \models p$ such that $I_k.\mathit{first} \models \varphi$ for each k. The loop terminates, if the split is finite and all I_k are. In this case the last state of I falsifies φ.

The semantics of calls $\rho(\underline{e}; \underline{z})$ for a declaration $\rho(\underline{x}; \underline{y})\ \{p\}$ unfolds to the procedure body p by rule (4.10). During the execution of p, the formal input parameters $\underline{x}$ are bound locally to their trace as a sequence of value vectors $\underline{\boldsymbol{a}}$ of which the first entry is initially given by the actual arguments $\underline{e}$. Likewise, the formal reference parameters $\underline{y}$ are bound to value vectors $\underline{\boldsymbol{b}}$, initially given by the actual ones $\underline{z}$. In contrast to $\underline{\boldsymbol{a}}$, the values $\underline{\boldsymbol{b}}$ of reference parameters become visible in the resulting interval in $\underline{z}$. Furthermore, the concatenation of intervals in the conclusion of the call rule is well defined, since the second s is collapsed with the first state of the subsequent interval and this state is fact equal to $I(\underline{z} \mapsto \underline{\boldsymbol{b}})$.

The **skip** steps (s, s) in (4.9) and (4.10) enforce that exiting a loop and procedure calls

Definition 4.2 (Nonatomic semantics of programs)**.** The nonatomic interval semantics $I \models p$ of programs is defined by the greatest fixpoint of the following inference system:

$$\frac{}{(s, s(\underline{x} \mapsto [\![\underline{e}]\!]_s)) \models \underline{x} := \underline{e}} \tag{4.1}$$

$$\frac{I \models p}{I \models p; q} \quad \text{if } I^{\uparrow} \tag{4.2}$$

$$\frac{I_1 \models p \qquad I_2 \models q}{I_1 \fatsemi I_2 \models p; q} \quad \text{if } I_1.\mathit{last} = I_2.\mathit{first} \tag{4.3}$$

$$\frac{I \models p}{I \models \mathtt{if}\ \varphi\ \mathtt{then}\ p\ \mathtt{else}\ q} \quad \text{if } I.\mathit{first} \models \varphi \tag{4.4}$$

$$\frac{I \models q}{I \models \mathtt{if}\ \varphi\ \mathtt{then}\ p\ \mathtt{else}\ q} \quad \text{if } I.\mathit{first} \not\models \varphi \tag{4.5}$$

$$\frac{I(\underline{x} \mapsto \underline{\boldsymbol{a}}) \models p}{I \models \mathtt{choose}\ \underline{x}\ \mathtt{with}\ \varphi\ \mathtt{in}\ p\ \mathtt{ifnone}\ q} \quad \text{if } I(\underline{x} \mapsto \underline{\boldsymbol{a}}).\mathit{first} \models \varphi \tag{4.6}$$

$$\frac{I \models q}{I \models \mathtt{choose}\ \underline{x}\ \mathtt{with}\ \varphi\ \mathtt{in}\ p\ \mathtt{ifnone}\ q} \quad \text{if } I.\mathit{first} \not\models \exists\, \underline{x}.\ \varphi \tag{4.7}$$

$$\frac{I \models p}{I \models \mathtt{while}\ \varphi\ \mathtt{do}\ p} \quad \text{if } I.\mathit{first} \models \varphi \text{ and } I^{\uparrow} \tag{4.8}$$

$$\frac{I_1 \models p \qquad I_2 \models \mathtt{while}\ \varphi\ \mathtt{do}\ p}{I_1 \fatsemi I_2 \models \mathtt{while}\ \varphi\ \mathtt{do}\ p} \quad \text{if } I_1.\mathit{first} \models \varphi \text{ and } I_1.\mathit{last} = I_2.\mathit{first}$$

$$\frac{}{(s, s) \models \mathtt{while}\ \varphi\ \mathtt{do}\ p} \quad \text{if } s \not\models \varphi \tag{4.9}$$

$$\frac{I(\underline{x}, \underline{y} \mapsto \underline{\boldsymbol{a}}, \underline{\boldsymbol{b}}) \models p}{(s, s) \fatsemi I(\underline{z} \mapsto \underline{\boldsymbol{b}}) \models \rho(\underline{e}; \underline{z})} \quad \begin{array}{l} \text{if } I.\mathit{first} = s \text{ and } \underline{\boldsymbol{a}} = ([\![\underline{e}]\!]_s, \cdots), \underline{\boldsymbol{b}} = (s(\underline{z}), \cdots) \\ \text{for a procedure definition } \rho(\underline{x}; \underline{y})\{p\} \\ \text{with formal inputs } \underline{x} \text{ outputs } \underline{y} \text{ and body } p \end{array} \tag{4.10}$$

take time, i.e., the rules are *productive*. A procedure that immediately calls itself in no time would not impose any constraints on the interval, rendering the semantics useless to observe any property of such a procedure, including nontermination. Nakata and Uustalu [115] describe some interesting anomalies that arise with non-productive rules.

Remark. In contrast to [55, Section 3.2], Definition 4.2 here has been simplified slightly: The fixpoint definition for the while loop is now part of the inference system instead of being defined by the greatest fixpoint operator ν explicitly. The simplification lends itself to define recursive procedure calls by the same mechanism—these had been omitted in [55].

Next, some interesting properties of the semantics are stated.

Proposition 4.3. *$I \models p$ implies that I has a step ($\#I \neq 0$).*

Proof. By structural induction on p. Every rule that does not make p smaller has indeed a proper step. □

Lemma 4.4 (Termination)**.** *The* least *fixpoint of Definition 4.2, written as $I \models_{\downarrow} p$ in the following, yields exactly the finite (terminating) executions of p:*

$$I \models_{\downarrow} p \quad \Longleftrightarrow \quad I \models p \text{ and } \#I \neq \infty.$$

Proof. Since the derivation rules are exactly the same on both sides of the equivalence, the only interesting part of the proof is which inductive argument to pick.

The first part of the $\Longrightarrow$ direction follows by the standard property that the least fixpoint is contained in the greatest one. Finiteness of I follows from the fact that the prerequisite $I \models_{\downarrow} p$ must have been derived by a finite number of inference steps and each step extends the interval only by finitely many states. The technical argument supporting this claim is an induction over the least fixpoint underlying $\models_{\downarrow}$, see [150].

The $\Longleftarrow$ direction can be shown by a lexicographic induction on the length of I, which is finite by assumption, and the structure of the program: either the interval is shortened (which is the case for entry into a while loop or a procedure call) or the premise refers to a structurally smaller program (which is the case for sequential composition, and conditionals). □

Proposition 4.5 (Existence of program executions)**.** *For each program p and initial state s there is some interval I with $I.first = s$ such that $I \models p$.*

Proof Sketch. The interval I is constructed incrementally, similarly to the proof for forward simulation of infinite intervals (Theorem 5.3). For a program p started in s', pick a *canonical* successor p' and the first step (s, s') from p to p', essentially by defining a deterministic structural operational semantics. Repeating this up to a counter k gives the k-th interval I_k of the diagonalization—I is chosen such that $I(k) := I_k.last$. Additional attention needs to be paid to the fact that p may terminate "prematurely" before the counter k is reached. The way how I was constructed shows that it satisfies indeed Definition 4.2. □

A similar approach is taken in [115], where an additional functional semantics is defined. The underlying definitional principle ensures that all functions are total so that some interval $I \models p$ exists as the return value of that function.

Note that to demonstrate a given instance of $I \models p$ (regardless of whether I is finite) is typically much simpler, as one can rely on the coinduction principle of the greatest fixpoint

underlying the definition of derivability. Dually to induction, such a claim follows from a set of assumptions when these can be propagated over some valid rule applications to the premises upwards the derivation tree (see e.g. the tutorial by Jacobs and Rutten [89] for a detailed explanation).

Example 4.6 (Coinduction)**.** Assume ($\star$) that $I = (s, s, \cdots)$ infinitely repeats an arbitrary single state s. We'll demonstrate that I models the execution of the trivial nonterminating loop by coinduction:

$$I \models \texttt{while true do skip}$$

One needs to find a suitable rule application so that each premise is either obvious and can be shown directly or has the same shape as the original statement. Of the three rules for **while** in Definition 4.2, the only applicable rule from Definition 4.2 is (4.2) since $s \models \texttt{true}$ (which rules out (4.9)) and any $I' \models \texttt{skip}$ for the body is always finite ($I^{\uparrow}$ does not hold, which rules out (4.8)). We have $I_1 = (s, s) \models \texttt{skip}$ in the first premise, and $I_2 = (s, \cdots)$ in the second premise has one state "less" than I such that $I_1 \fatsemi I_2 = I$. The remainder I_2 satisfies the original assumption ($\star$) to repeat s infinitely often and therefore the second premise is covered by the coinductive hypothesis. ■

4.2 Data-Type like Abstract State Machines

This section formalizes data type like ASMs that provide a sequential interface to the world in terms of operations, which in turn are defined by programs. The nonatomic program semantics $I \models p$ is first lifted to an atomic relational one $[\![p]\!]$ that collapses finite intervals into their first and last states and produces $\bot$ as successor state for infinite intervals. Based on this the (atomic) semantics of operations $[\![\texttt{Op}]\!]$ additionally checks preconditions and copies inputs and outputs from and back to the caller, so that $[\![\texttt{Op}]\!]$ finally provides the transitions of the machine.

Definition 4.7 (Atomic semantics of programs)**.** The relational atomic semantics of programs $[\![p]\!] \subseteq St \times St$ collapses the nonatomic interval semantics $I \models p$ into their first and last states (first line). In contrast to the interval semantics of Definition 4.2, nontermination is expressible by final states $\bot$ (assuming $\bot \in St$), corresponding to infinite intervals (second line). A program started in a $\bot$ state "after" a nonterminating computation does nothing (last line), coinciding with the absorption of I_2 in the composition $I_1 \fatsemi I_2 = I_1$ when I_1 is infinite.

$$\begin{aligned}
(s, s') \in [\![p]\!] \quad &\iff \quad (s, \cdots, s') \models p \text{ for a finite interval, possibly ending in } \bot \\
(s, \bot) \in [\![p]\!] \quad &\iff \quad (s, \cdots) \models p \text{ for an infinite interval} \\
(\bot, \bot) \in [\![p]\!] \quad &\text{always}
\end{aligned}$$

Definition 4.8 (Atomic semantics of operations)**.** The atomic semantics of operations $\texttt{Op} = (\texttt{pre}, in, p, out)$ is based on the corresponding atomic semantics $[\![p]\!]$ of the program p. Additionally, the precondition is evaluated and if it does not hold, the result is unspecified in agreement with the interpretation as a contract, which is modeled by a successor

state $\bot$.

$$(i, s, s', o) \in [\![\mathtt{Op}]\!] \iff \begin{cases} (s(in \mapsto i), s') \in [\![p]\!] \text{ and } o = s'(out), & s(in \mapsto i) \models \mathtt{pre} \\ s' = \bot \text{ and } o \text{ arbitrary}, & \text{otherwise.} \end{cases}$$

The actual inputs values i are initially copied to the state in which the body p starts to execute. Upon termination, the output is ready in out. The output $s'(out)$ is unspecified and irrelevant when $s' = \bot$.

Remark. In [55] we have permitted an arbitrary state $s' \in St$ when the precondition is violated. These extra outcomes are unnecessary, because these are only possibly needed in the refinement proofs, however the worst possible result $\bot$ already matches any concrete behavior. The technical consideration is that characterizing (5.3) of context steps context becomes a little simpler with the present approach.

The semantics of ASMs is interpreted in terms of labeled transition systems. These are a natural choice to encode the behavior of data type like machines: the transitions correspond to invocations of operations $\mathtt{Op}_j$, whereas the labels attached to the transitions expose the index j alongside the input passed to the operation as well as the output returned.

Definition 4.9 (Labeled transition system)**.** A transition system $T = (S, L, Init, \longrightarrow)$ over a state space S and a set of labels L has a set of initial states $Init \subseteq St$ and a transition relation $\longrightarrow \subseteq S \times L \times S$ between pairs of states and a label. An individual transition from state $s \in S$ to $s' \in S$ labelled by $l \in L$ is written as $s \xrightarrow{l} s'$.

Definition 4.10 (Executions and runs)**.** An execution of a transition system T exhibiting sequences of observations $\underline{l} \in L^\omega$ is an interval I over the corresponding state space S, written $I \in execs^T(\underline{l})$, such that $\#I = |\underline{l}|$ and

$$I(k) \xrightarrow{\underline{l}(k)} I(k+1) \qquad \text{for all } k < \#I.$$

It is a run, written $I \in runs^T(\underline{l})$, if $I.first \in Init$, i.e., the first state is an initial one.

From now on, we'll use the specific instance that adds a distinguished element $\bot$ to the base set of states $St := S \uplus \{\bot\}$ to denote divergence, and labels $l = j, i, o$ are triples of an index of an operation j, and input value i and an output value o. We will leave the set of labels L implicit in the following but assume that it contains triples j, i, o for all $j \in \mathcal{J}$ and possible types of inputs and outputs of the machines in question. We require that a system cannot recover from divergence, i.e., $\bot \xrightarrow{\cdots} s'$ implies $s' = \bot$, too (which is satisfied by Definition 4.8).

Definition 4.11 (Semantics of data type like ASMs)**.** The semantics of a data type like ASM $\mathtt{M} = (\underline{x}\colon \underline{\mathtt{St}}, \mathtt{Init}, \{\mathtt{Op}_j\}_{j\in\mathcal{J}})$ is given by the transition system $M = (St, Init, \longrightarrow)$ subject to the conventions above. The state space $St := A_{\underline{\mathtt{St}}} \uplus \{\bot\}$ is instantiated by the carrier sorts of the algebraic types $\mathtt{St}$ of the state variables $\underline{x}$. The initial states of M are the ones satisfying the initialization condition $Init = \{s \in St \mid s \models \mathtt{Init}\}$, and the next state relation is instantiated by the atomic semantics of the called operation $\mathtt{Op}_j$, which provides the observed inputs i and outputs o.

$$s \xrightarrow{j,i,o} s' \iff (i, s, s', o) \in [\![\mathtt{Op}_j]\!].$$

Executions $execs^{\mathsf{M}}$ and runs $runs^{\mathsf{M}}$ are inherited from the underlying transition system.

Remark. A data type ASM can be encoded by a conventional ASM with a main program consisting of a big choice between guarded fragments for each $\mathtt{Op}_j = (\mathtt{pre}_j, in_j, out)$ of the form

$$\textbf{if } \mathtt{op} = j \textbf{ then } \{\textbf{if } \mathtt{pre}_j \textbf{ then } p_j \textbf{ else abort}\}$$

The index of the called operation is provided within an "input location" `op`, and similarly the input in a location in_j which can be read by by p_j. Complementary, the output is passed back to the environment in the "output location" out_j, which p_j is supposed to assign (cf. [29]). This syntactic embedding agrees with the semantics of Definitions 4.7 and 4.8, as $(s, s') \in [\![\textbf{abort}]\!]$ only if $s' = \bot$.

Example 4.12 (A Mini File System)**.** This example introduces an idealistic model of a file system as a data type like ASM, referred to as Mini FS in the following.

Purpose of the Mini FS is to store a set of *files* which are referenced by *names*. It provides four operations to create and delete files, and to read and write their content. It can be seen as an abstraction of a POSIX file store with a flat namespace (there are no directories) where the content of files is left abstract (they are read and written in their entirety).

Mini FS is specified as a data type like ASM with one state variable $fs\colon \mathtt{Name} \rightharpoonup \mathtt{File}$ that is a finite map from abstract names of type `Name` to abstract files of type `File` that is initially empty, i.e., $\mathtt{Init}(fs) \coloneqq (fs = \varnothing)$. Figure 4.2 shows the four operations of the model, i.e., the set of indices is $\mathcal{J} = \{\mathtt{create}, \mathtt{delete}, \mathtt{read}, \mathtt{write}\}$, although in concrete ASM listings, the operations are indicated by their name instead of the indexed notation $\mathtt{Op}_j$ for some $j \in \mathcal{J}$. Each operation is equipped with a precondition—indicated by an explicit keyword—that specifies the conditions when it is appropriate to call the respective operation.

The `create` operation allocates a new file with an empty content, denoted by the constant `empty`: `File`. It is required that the name doesn't already exist in the domain of *fs*. The `delete` operation conversely removes the name from the mapping. The `read` operation returns the content associated with the name given as input parameter *name* in the output parameter *file*. The write operation stores new content *file* under the given *name*, where the assignment $fs[name] \coloneqq file$ abbreviates $fs \coloneqq fs[name \mapsto file]$ that overrides the previous value at the key *name* with the new value *file*.

In concrete ASM listings as the one in Figure 4.2, inputs are separated by a semicolon from the outputs by convention, i.e., the listing in Figure 4.2 stands for

$$\mathtt{Op}_{\mathtt{read}} = (\underbrace{name \in \mathtt{dom}(fs)}_{\textit{precondition}}, \underbrace{name}_{\textit{input}}, \underbrace{file \coloneqq fs[name]}_{\textit{program}}, \underbrace{file}_{\textit{output}})$$

in the notation of Definition 4.1.

The example demonstrates how ASM operations are broken down to algebraic ones, here lookup and override on functional maps as described in Section 3.1. This pattern is fairly common, although the real models of the case study are typically much more complex and include for instance case distinctions, multiple assignments, and calls to operations of subcomponents.

Note that except for `read` the preconditions given are stricter than necessary to just guarantee that the algebraic operations are well-defined (for instance, re-creation of a file

```
create(name)
  precondition ¬ name ∈ dom(fs)
  fs[name] := empty

delete(name)
  precondition name ∈ dom(fs)
  fs := fs − name

read(name; file)
  precondition name ∈ dom(fs)
  file := fs[name]

write(name, file)
  precondition name ∈ dom(fs)
  fs[name] := file
```

Figure 4.2: Specification of Mini FS, an idealized file system. The state variable *fs* is omitted from the parameter lists.

under an existing name could simply reset the contents to `empty`). This is to document specific design decisions and intended use of the interface. Such stronger preconditions become relevant when a specification is refined towards an implementation with a different representation of the state, which may have less liberal well-definedness criteria (e.g., re-creating an existing file could otherwise lead to undefined effects or a violation of internal invariants).

4.3 Submachine Composition

This section clarifies interface use $\mathtt{M}(\_)$ (corresponding to the pictographic notation $\mathtt{M}$—⊂) in terms of "filling the hole" in the context $\mathtt{M}$ by a submachine $\mathtt{X}$ to yield the composition $\mathtt{M}(\mathtt{X})$. As already outlined, interface provision is expressed by operations $\mathtt{Op}_j^{\mathtt{X}}$ superscripted by the name of the machine. Complementary, interface use is expressed by calls to such operations that are now admitted in the grammar of programs by extending Definition 3.2 with a new clause

$$p \quad ::= \quad \cdots \mid \mathtt{Op}^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$$

with actual input parameters $\underline{e}$ and output parameters $\underline{z}$. The state of $\underline{xx}$ of $\mathtt{X}$ is passed as an explicit reference parameter, to avoid global variables in the calculus. Submachine composition $\mathtt{M}(\mathtt{X})$ is built from a context or supermachine with a hole

$$\mathtt{M}(\_) = (\underline{mx}\colon \mathit{St}^{\mathtt{M}}, \mathtt{Init}^{\mathtt{M}}, \{\mathtt{Op}_k^{\mathtt{M}}\}_{k\in K})$$

where the programs occurring as the bodies of the operations $\mathtt{Op}_k^{\mathtt{M}}$ may now contain such calls $\mathtt{Op}^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$ (for a specific $\mathtt{X}$). Filling the hole with the required submachine

$$\mathtt{X} = (\underline{xx}\colon \mathit{St}^{\mathtt{X}}, \mathtt{Init}^{\mathtt{X}}, \{\mathtt{Op}_j^{\mathtt{X}}\}_{j\in J}).$$

yields the composed system $\mathtt{M}(\mathtt{X})$, which is again a proper machine. In order for the composition to be well-defined, the state variables $\underline{mx}$ and $\underline{xx}$ have to be disjoint. Furthermore, the signature of the operations provided by $\mathtt{X}$ must match the expectations of $\mathtt{M}(\_)$.

Definition 4.13 (Submachine composition)**.** When the composition is well-defined, it gives rise to the following machine

$$\mathtt{M}(\mathtt{X}) := (\underline{mx}, \underline{xx}\colon \mathit{St}^{\mathtt{M}}, \mathit{St}^{\mathtt{X}}, \quad \mathtt{Init}^{\mathtt{M}} \wedge \mathtt{Init}^{\mathtt{X}}, \quad \{\mathtt{Op}_k^{\mathtt{M}}\}_{k\in K})$$

that combines the state space and initialization and exposes the operations of $\mathtt{M}$.

Although the programs of the operations $\mathtt{Op}^{\mathtt{M}}$ may access the state of $\mathtt{X}$ only indirectly through the interface of $\mathtt{X}$, the invariants and preconditions of $\mathtt{M}$ may well refer to the internal state $\underline{xx}$ of the submachine. The reason is simply that invariants and preconditions appear only in the verification but not in the running code of a composed system—the verification of $\mathtt{M}(\mathtt{X})$ needs to render $\mathtt{X}$ transparent anyway.

The states s of a well-defined compound $\mathtt{M}(\mathtt{X})$ can be split disjointly into $xs = s(\underline{xx})$ and $ms = s(\underline{mx})$, which is written $s = xs \oplus ms$ in the following. Likewise, intervals I can be split point wise, separating the two states into two sub-intervals as $I^{\mathtt{X}} = I(\underline{xx})$ and $I^{\mathtt{M}} = I(\underline{mx})$ such that $I = I^{\mathtt{X}} \oplus I^{\mathtt{M}}$.

Like calls to named procedures $\rho(\underline{e}; \underline{z})$, submachine calls $\mathtt{Op}^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$ accept actual argument expressions $\underline{e}$ for their input parameters and write their result to actual output parameters $\underline{z}$. Semantically, however, submachine calls are handled differently than named procedures calls. While the latter are just unfolded to their body according to Definition 4.2, exposing their internal steps, submachine calls are integrated *compositionally* by referring to the atomic semantics $[\![\mathtt{Op}_j^{\mathtt{X}}]\!]$ of the operation, which decouples the context from the details how such operations are realized.

Definition 4.14 (Calls to operations of submachines)**.** A call to an operation $\mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$ of a machine $\mathtt{X} = (\underline{xx}\colon \mathtt{St}^{\mathtt{X}}, \mathtt{Init}^{\mathtt{X}}, \{\mathtt{Op}_j^{\mathtt{X}}\}_{j\in\mathcal{J}})$ with actual arguments $\underline{e}$ and $\underline{z}$ from a context machine $\mathtt{M}$ generates a one-step interval I when the call terminates and a potentially infinite interval continuation with only $\bot$ states for a diverging call

$$\frac{}{(xs \oplus ms, xs' \oplus ms(\underline{z} \mapsto o)) \models \mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})} \quad \text{if } (i, xs, xs', o) \in [\![\mathtt{Op}_j^{\mathtt{X}}]\!] \text{ and } xs' \neq \bot$$

$$\frac{}{(xs \oplus ms, \bot, \bot, \cdots) \models \mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})} \quad \text{if } (i, xs, \bot, o) \in [\![\mathtt{Op}_j^{\mathtt{X}}]\!]$$

where $i = [\![\underline{e}]\!](ms)$ in both cases and o is effectively unconstrained in the second rule by the semantics of diverging operations.

The first rule captures terminating calls, whereas the second rule captures diverging calls. The inputs $[\![\underline{e}]\!](ms)$ are evaluated from the outer state ms of $\mathtt{M}$ and passed to the atomic semantic of $\mathtt{Op}^{\mathtt{X}}$; conversely, the output o is passed back to the state of $\mathtt{X}$ by $ms(\underline{z} \mapsto o)$ in the first rule.

Operation calls conform to the information hiding principles of proper use of the submachine $\mathtt{X}$: The actual parameters $\underline{e}$, $\underline{z}$ may reference only the state of the supermachine (reflected by the evaluation of the input $i = [\![\underline{e}]\!](ms)$ in the state of $\mathtt{M}$), whereas the successor $\mathtt{M}$ state is unchanged except for storing the actual output in $\underline{z}$. Note that there is no premise that unfolds the body of $\mathtt{Op}_j^{\mathtt{X}}$, reflecting the fact that there is a semantic dependence on $\mathtt{Op}_j^{\mathtt{X}}$ only.

For a diverging submachine call the execution continues with an arbitrary number of $\bot$ states. This reflects the fact that nothing is known "after" the diverging call. For technical reasons that will be explained in Section 5.3 it is convenient not to fix the length of the outcome, effectively permitting any refined behavior of an implementation of $\mathtt{Op}_j^{\mathtt{X}}$ when $\mathtt{X}$ is a specification level machine. In fact, arbitrary $\bot$-states can be appended to an interval $I \models p$ that already ends with $\bot$ (shown as Lemma 5.8 later on).

Example 4.15 (A Client of the Mini File System)**.** Consider a client shown in Figure 4.3 of the Mini FS from Example 4.12. The task is to copy a file denoted by *from*: *Name* to a

```
copy(from, to)
  precondition from ∈ dom(fs) ∧ to ∉ dom(fs)
  let file = ? in
    read(from; file)
    create(to)
    write(to, file)
  let file′ = ? in
    read(to; file′)
    assert file = file′
```

Sequence of assignments
as executed by the submachine

```
file := fs[from]
fs[to] := empty
fs[to] := file

file′ := fs[to]
```

Figure 4.3: Copying a file with the Mini FS interface.

new name *to*: `Name`, using the operations `create`, `read`, and `write` provided by the Mini FS machine.

The precondition just states that *from* exists and *to* does not (refering to the state *fs* of the submachine). The subsequent line introduces a local variable *file* to temporarily hold the contents to be copied, it is passed as an output parameter of `read` (following the semicolon) and assigned to $fs[from]$ as a result of the call as shown in the listing on the right in Figure 4.3 (cf. the definition of `read` in Example 4.12). The next two lines create the destination *to* and write the contents such that $fs[to] = file$ afterwards (cf. the definition of `write`).

After copying the file, the client actually verifies that the operation succeeded by reading the contents of *to* into another local variable *file*′, which is then compared to the original one.[1]

From the sequence of assignments executed by the submachine as shown at the right in Figure 4.3, it is straight forward to conclude that $file = fs[from]$ equals $file' = fs[to]$ when the assertion is evaluated. The next section shows how to verify that in the calculus the assertion holds when the precondition of `copy` is satisfied (Example 4.18).

4.4 Calculus

The three weakest-precondition modalities from Section 3.3 can be defined as follows by the nonatomic semantics of programs

$$\begin{array}{lcll} s \models [p]\,\varphi & \iff & \text{for all } s' \text{ with } (s,s') \in [\![p]\!]\text{:} & s' \neq \bot \text{ implies } s' \models \varphi \\ s \models \langle p \rangle\,\varphi & \iff & \text{for some } s' \text{ with } (s,s') \in [\![p]\!]\text{:} & s' \neq \bot \text{ and } s' \models \varphi \\ s \models \langle\!|p|\!\rangle\,\varphi & \iff & \text{for all } s' \text{ with } (s,s') \in [\![p]\!]\text{:} & s' \neq \bot \text{ and } s' \models \varphi \end{array}$$

so that standard symbolic execution rules and equivalences hold (for example the termination splitting lemma $\langle\!|p|\!\rangle\varphi \leftrightarrow \langle\!|p|\!\rangle\texttt{true} \wedge [p]\varphi$ and the duality $\langle p\rangle\neg\varphi \leftrightarrow \neg[p]\varphi$).

Two symbolic execution rules support proofs about operations $\texttt{Op} = (\texttt{pre}, \underline{in}, p, \underline{out})$ of data-type like ASMs. In the sequent calculus, total correctness

$$\Gamma \vdash \langle\!|\texttt{Op}(\underline{e}; \underline{x}, \underline{z})|\!\rangle\ \texttt{post}$$

states that when the operation `Op` computing on $\underline{x}$ is called with actual inputs $\underline{e}$ and actual outputs $\underline{z}$ in a state satisfying assumptions Γ (a list of formulas), it will terminate and the final state satisfies a postcondition `post` (cf. Section 3.3).

[1] **assert** $\varphi \equiv$ **if** $\neg\varphi$ **then abort**.

The first rule simply inlines the body of the operation p:

$$\frac{\Gamma, \underline{in} = \underline{e} \vdash \texttt{pre} \qquad \Gamma, \underline{in} = \underline{e}, \texttt{pre} \vdash \langle\!|p^{\underline{z}}_{\underline{y}}|\!\rangle\ \texttt{post}}{\Gamma \vdash \langle\!|\texttt{Op}(\underline{e}; \underline{x}, \underline{z})|\!\rangle\ \texttt{post}}\ \text{call} \tag{4.11}$$

The first premise checks whether the precondition is established by Γ when the formal parameters $\underline{in}$ are assigned to the actual ones $\underline{e}$ (some renaming would be required when the variable $\underline{in}$ happens to be free in Γ). The second premise reduces correctness of the operation to the correctness of program p provided the precondition holds (where the formal output parameters $\underline{out}$ are replaced to refer to the actual ones $\underline{z}$). The rule the existential modality $\langle\_\rangle\_$ is alike. The rule for partial correctness omits the first premise.

The second rule used to prove properties about operations permits one to use a lemma, a correctness assertion of the form $\varphi \vdash \langle\!|\texttt{Op}(\underline{in}; \underline{x}, \underline{out})|\!\rangle\ \psi$, to dispatch the call without unfolding it:

$$\frac{\Gamma, \underline{in} = \underline{e} \vdash \varphi \qquad \Gamma, \underline{in} = \underline{e}, \varphi \vdash \forall\, \underline{z}.\ \psi^{\underline{z}}_{\underline{out}} \rightarrow \texttt{post}}{\Gamma \vdash \langle\!|\texttt{Op}(\underline{e}; \underline{x}, \underline{z})|\!\rangle\ \texttt{post}}\ \text{lemma}$$

This time, the first premise asserts the precondition of the lemma (which in turn must imply the precondition of the operation for the lemma to hold). The second premise replaces the call by the postcondition ψ of the lemma, adapted analogously to the call rule for the actual reference parameters. The universal quantifier prevents any information in Γ about the initial value of $\underline{z}$ to leak into the final state.

Variations of this rule for the possible combinations of different modalities are implemented in the KIV system and there is an option to keep the original execution of `Op` around when the lemma is itself is insufficient to establish `post`. Detailed information about these proof rules and how they are automated can be found in: *A Practical Course on KIV* [142].

Often, it makes sense to constrain the states of transition systems resp. machines by *invariants* that must hold throughout every run.

Definition 4.16 (Invariant)**.** A property $Inv \subseteq St$ is an invariant of a transition system T, if for every $I \in runs^T(\underline{j}, \underline{i}, \underline{o})$ we have $I(k) \in Inv$ for every k.

We give the following standard conditions without proof.

Proposition 4.17. *An invariant Inv can be proved inductively with*

$$Init \subseteq Inv \quad \text{and} \quad Inv \fatsemi (\xrightarrow{j,i,o}) \subseteq Inv,$$

that is, the invariant is established in initial states and propagated over each transition. The corresponding syntactic proof obligations that establishes $Inv = [\![\texttt{Inv}]\!]$ in the calculus are straight-forward, when $\texttt{Inv}(\underline{x})$ is given by a predicate over some state variables $\underline{x}$.

$\texttt{Init}(\underline{x}) \vdash \texttt{Inv}(\underline{x})$	initialization
$\texttt{Inv}(\underline{x}), \texttt{pre}_j(in, \underline{x}) \vdash \langle\!|\texttt{Op}_j(in; \underline{xx}, out)|\!\rangle\ \texttt{Inv}(\underline{x})$	propagation

Besides deduction over sequential programs the KIV system provides a symbolic execution calculus for concurrent programs as well. It is based on the interval temporal logic RGITL [145] that is compatible with the semantics shown in Section 4.1 (for an empty

environment). Formulas in the logic are evaluated over an entire interval instead of just a single state, for example $I \models \Box\varphi$ requires φ to hold in all suffixes of I. Correctness assertions in RGITL are expressed as sequents $[p]_{\underline{x}} \vdash \varphi$ for a program p computing on variables $\underline{x}$, meaning that $I \models \varphi$ should hold for every execution $I \models p$ (in the absence of other concurrent processes). The connection to weakest precondition calculus is given by

$$\langle\!|p|\!\rangle\ \varphi \iff [p]_{\underline{x}} \vdash \Diamond(\mathbf{last} \wedge \varphi) \qquad \text{and} \qquad [p]\ \varphi \iff [p]_{\underline{x}} \vdash \Box(\mathbf{last} \to \varphi),$$

where the atomic formula **last** holds only at the end of an interval. It is thereby possible to switch to the more expressive temporal calculus for properties that must refer to intermediate states as it is the case with power cut safety.

Beyond crash safety, the tight integration of the different formalisms permits to extend the verification to a concurrent setting in the future, such that the Flashix file system can satisfy multiple client applications running in an interleaved fashion at the same time; and similarly internal operations (such as wear-leveling and garbage collection) can be executed in parallel to the ordinary ones in the background. For such proofs, process local reasoning in terms of rely-/guarantee decomposition is provided [152].

Example 4.18 (Verification of `copy`)**.** Let's prove that the assertion in the `copy` operation of the client shown in Example 4.15 holds, i.e., that a call to `copy` terminates normally. The proof is presented as a series of intermediate proof goals, not as a complete derivation tree. We start from the conclusion

$$\mathit{from} \neq \mathit{to} \vdash \langle\!|\texttt{copy}(\mathit{from}, \mathit{to})|\!\rangle\ \texttt{true},$$

where $\mathit{from} \neq \mathit{to}$ corresponds to the precondition of `copy`. We can inline the call to `copy` by rule (4.11), yielding

$$\mathit{from} \neq \mathit{to} \vdash \langle\!|\texttt{read}(\mathit{from}; \mathit{file}); \texttt{create}(\mathit{to}); \texttt{write}(\mathit{to}, \mathit{file}), \ldots |\!\rangle\ \texttt{true}$$

and subsequently call the submachine procedures and execute the assignment by rule (3.1), yielding for fresh fs', fs'' the sequent

$$\begin{aligned}&\mathit{from} \neq \mathit{to}, \mathit{file} = \mathit{fs}[\mathit{from}], \mathit{fs}' = \mathit{fs}[\mathit{to} \mapsto \texttt{empty}], \mathit{fs}'' = \mathit{fs}'[\mathit{to} \mapsto \mathit{file}]\\ &\vdash \langle\!|\texttt{read}(\mathit{to}; \mathit{file}'); \mathbf{assert}\ \mathit{file} = \mathit{file}'|\!\rangle\ \texttt{true}.\end{aligned}$$

The next read returns $\mathit{file}' = \mathit{fs}''[\mathit{to}]$ and we symbolically execute the assertion via the equivalence $(\langle\!|\mathbf{assert}\ \varphi|\!\rangle\ \texttt{true}) \leftrightarrow \varphi$. Substituting the equations into the conclusion yields

$$\begin{aligned}&\mathit{from} \neq \mathit{to}, \mathit{file} = \mathit{fs}[\mathit{from}]\\ &\vdash \mathit{fs}[\mathit{to} \mapsto \texttt{empty}][\mathit{to} \mapsto \mathit{file}][\mathit{to}] = \mathit{fs}[\mathit{to} \mapsto \texttt{empty}][\mathit{to} \mapsto \mathit{file}][\mathit{from}]\end{aligned}$$

which reduces to the trivial goal $\vdash \mathit{file} = \mathit{file}$, because *from* and *to* are different from the precondition of `copy`. ■

4.5 Extracting Submachine Runs

This section discusses how the steps of a program of a composed machine `M(X)` are related to the executions and runs of the submachine `X`. More specifically, given $I^{\mathsf{X}} \oplus I^{\mathsf{M}} \models p$,

we will establish that $I^{\mathtt{X}} \in \mathit{execs}^{\mathtt{X}}_{\tau}(\underline{j}, \underline{i}, \underline{o})$ for some observed call sequence $\underline{j}$, inputs $\underline{i}$, and outputs $\underline{o}$ that reflect the interaction of the two machines over the interface.

Hence, the dual role of intervals as outlined at the start of this chapter is made precise (cf. Figure 4.1). The relation between a context and its submachine later serves as the basis for a substitution theorem as motivated in the backround on refinement in Section 3.4.

Intuitively, the execution of the program exhibits steps that are either regular assignments, which modify the state ms of $\mathtt{M}$ to some ms', but not the state xs of $\mathtt{X}$, which is encapsulated, or we have a submachine call $\mathtt{Op}^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$ such that $(i, xs, xs', o) \in [\![\mathtt{Op}^{\mathtt{X}}_j]\!]$ for $i = [\![\underline{e}]\!](ms)$ and $ms' = ms(\underline{z} \mapsto o)$. In principle, these two conditions characterize warrant that $I^{\mathtt{X}}$ with such steps $I^{\mathtt{X}}(k) = xs$ and $I^{\mathtt{X}}(k+1) = xs'$ is an execution of $\mathtt{X}$. However, there are two aspects that complicate the matter:

1. Regular assignments in p induce steps that are not submachine calls. These need to be mapped to "stutter" steps in $I^{\mathtt{X}}$ (indicated by the subscript τ of *execs*), which leave the state of $\mathtt{X}$ unchanged and are there just to align the lengths of $I^{\mathtt{X}}$ and $I^{\mathtt{M}}$.
2. Unfortunately, a given interval $I^{\mathtt{X}} \oplus I^{\mathtt{M}} \models p$ does not fully determine the outcomes of nondeterministic choices made by the execution when viewed in *retrospect*: There may be different sequences $\underline{j}$ matching the observed states found in $I^{\mathtt{X}}$, as demonstrated shortly in Example 4.20. The solution is to annotate the interval semantics by the observed call sequence (and in/outputs) as in $I^{\mathtt{X}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p$.

The first aspect is addressed by extending the set of indices as $\mathcal{J}_\tau = \mathcal{J} \uplus \{\tau\}$ to include an extra marker τ indicates stuttering transitions that. Such steps can be used to model inactivity or explicit waiting for a context.

Definition 4.19 (Stuttering executions and runs)**.** The stuttering executions of a transition system T, written with subscript τ as $I \in \mathit{execs}^{T}_{\tau}(\underline{j}, \underline{i}, \underline{o})$ and likewise runs $I \in \mathit{runs}^{T}_{\tau}(\underline{j}, \underline{i}, \underline{o})$, are defined by steps

$$I(k) = I(k+1), \quad \text{if } \underline{j}(k) = \tau \ \ (\text{and } \underline{i}(k), \underline{o}(k) \text{ arbitrary})$$
$$I(k) \xrightarrow{\underline{j}(k), \underline{i}(k), \underline{o}(k)} I(k+1), \quad \text{otherwise.}$$

The weakness of the interval semantics to precisely resolve nondeterministic decisions in retrospect is illustrated by the following example.

Example 4.20. Consider the simple program $p \equiv \mathtt{Op}^{\mathtt{X}}_j()$ where the operation $\mathtt{Op}^{\mathtt{X}}_j()\{\ \mathtt{skip}\ \}$ just skips. An interval $I^{\mathtt{X}} \models p$ such that $I^{\mathtt{X}} = (xs, xs)$ is an execution fragment $I^{\mathtt{X}} \in \mathit{execs}^{\mathtt{X}}_{\tau}(j)$ for a one-element call sequence $\underline{j} = (j)$ consisting just of the index j. However, $I^{\mathtt{X}} \in \mathit{execs}^{\mathtt{X}}_{\tau}(\tau)$ for the call sequence $\underline{j} = (\tau)$ is valid, too.

Example 4.20 is deliberately kept simple. It wouldn't be too hard to construct a more realistic example (in particular one that does not depend on τ steps). To expose the weakness, it suffices that the call sequence, inputs and outputs are not uniquely determined by the interval $I^{\mathtt{X}}$ alone.

The solution is to strengthen the nonatomic semantics to judgements $I, \underline{j}, \underline{i}, \underline{o} \models p$, where the additional sequences record the values of the respective call steps. Submachine calls $\mathtt{Op}^{\mathtt{C}}_j(\underline{e}; \underline{cx}, \underline{z})$ determine exactly one element of these sequences, in particular $\underline{j} = (j)$ (see below). For assignments in the program of $\mathtt{M}$, $\underline{j} = (\tau)$ gives a stutter step for $\mathtt{A}$ and $\underline{i}$ resp. $\underline{o}$ are arbitrary one-element sequences. Furthermore, the three sequences are threaded through sequential composition just like the interval. The problem in Example 4.20 is ruled out by the fact that $I^{\mathtt{C}}, \underline{j}, \ldots \models \mathtt{Op}^{\mathtt{C}}_j()$ fixes $\underline{j} = (j) \neq (\tau)$.

Definition 4.21 (Nonatomic semantics revisited)**.** The nonatomic semantics of programs that uniquely determines the call sequence, inputs, and outputs of the submachine derives judgements of the form $I, \underline{j}, \underline{i}, \underline{o} \models p$. The interesting rules are

$$\frac{}{(s, s'), (\tau), (i), (o) \models \underline{x} := \underline{e}} \text{ where } s' = s(\underline{x} \mapsto [\![\underline{e}]\!]_s)$$

$$\frac{I, \underline{j}, \underline{i}, \underline{o} \models p}{I, \underline{j}, \underline{i}, \underline{o} \models p; q}\ I^{\uparrow}$$

$$\frac{I_1, \underline{j}_1, \underline{i}_1, \underline{o}_1 \models p \qquad I_2, \underline{j}_2, \underline{i}_2, \underline{o}_2 \models q}{I_1 \fatsemi I_2,\ \underline{j}_1\underline{j}_2,\ \underline{i}_1\underline{i}_2,\ \underline{o}_1\underline{o}_2 \models p; q}\ I_1.last = I_2.first$$

$$\frac{}{(xs \oplus ms, xs' \oplus ms(\underline{z} \mapsto o)), (j), (i), (o) \models \mathtt{Op}_j^{\mathsf{X}}(\underline{e}; \underline{xx}, \underline{z})}\ xs' \neq \bot$$

$$\frac{}{(xs \oplus ms, \bot, \bot, \cdots), (j, ..), (i, ..), (o, ..) \models \mathtt{Op}_j^{\mathsf{X}}(\underline{e}; \underline{xx}, \underline{z})}\ xs' = \bot$$

where $(i, xs, xs', o) \in [\![\mathtt{Op}_j^{\mathsf{X}}]\!]$ and $i = [\![\underline{e}]\!](ms)$ as in Definition 4.14. Note that a diverging submachine call admits an arbitrary call sequence and inputs/outputs (which of course must have the same length as the interval). The cases for the other programming constructs are omitted here, because they do not add new aspects: the rules for conditionals and nondeterministic choice just propagate the sequences to the respective premises, whereas loops and calls to named procedures resemble specific instances of the sequential composition.

Lemma 4.22 (Submachine execution)**.** *From a program p over X we can get the call sequence, inputs, and outputs explicitly as*

$$I^{\mathsf{X}} \oplus I^{\mathsf{M}} \models p \quad \Longleftrightarrow \quad \exists\, \underline{j}, \underline{i}, \underline{o}.\ I^{\mathsf{X}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p$$

so that the additional information $\underline{j}, \underline{i}, \underline{o}$ from the existential induces the corresponding stuttering execution fragment of X

$$I^{\mathsf{X}} \in execs_{\tau}^{\mathsf{X}}(\underline{j}, \underline{i}, \underline{o}),$$

where the τ-transitions correspond to the steps performed by p that are not calls.

In the following a simple fact distributes modification of a compound state and interval down to the respective component, provided $\underline{y}$ is part of the context.

$$\begin{aligned} (xs \oplus ms)(\underline{y} \mapsto \underline{a}) &= xs \oplus (ms(\underline{y} \mapsto \underline{a})) \\ (I^{\mathsf{X}} \oplus I^{\mathsf{M}})(\underline{y} \mapsto \underline{\boldsymbol{a}}) &= I^{\mathsf{X}} \oplus (I^{\mathsf{M}}(\underline{y} \mapsto \underline{\boldsymbol{a}})) \end{aligned} \tag{4.12}$$

Proof of Lemma 4.22. The $\Longrightarrow$ direction of the first formula constructs $\underline{j}, \underline{i}, \underline{o}$ incrementally, using the approach from Proposition 4.5 (not detailed here). The $\Longleftarrow$ direction constructions a derivation of $I^{\mathsf{X}} \oplus I^{\mathsf{M}} \models p$ from $I^{\mathsf{X}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p$ for given, arbitrary $\underline{j}, \underline{i}, \underline{o}$ by the coinduction principle of the greatest fixpoint. The correspondence is immediate by comparing the individual rules side by side.

To show that I^{X} is indeed an execution for the given calls, inputs, and outputs, all steps $k < \#I^{\mathsf{X}}$ must satisfy

$$I^{\mathsf{X}}(k) = I^{\mathsf{X}}(k+1), \qquad \text{when } \underline{j}(k) = \tau \tag{4.13}$$

$$(\underline{i}(k), I^{\mathsf{X}}(k), I^{\mathsf{X}}(k+1), \underline{o}(k)) \in [\![\mathtt{Op}^{\mathsf{X}}_{\underline{j}(k)}]\!], \qquad \text{otherwise.} \tag{4.14}$$

Assuming not all steps satisfy these conditions, then pick k as the first offending one. A lexicographic induction over this k and the structure of p leads to a contradiction—step k is a "good" one and the assumption is false. An analysis over the derivation of the provided $I^{\mathsf{X}}, \underline{j}, \underline{i}, \underline{o} \models p$ gives the following cases:

- An assignment implies that I^{X} has one step and consequently $k = 0$ from the assumption $k < \#I^{\mathsf{X}}$. Then (4.13) follows from the enriched semantics in Definition 4.21 and (4.12). A submachine call is analogous, except that (4.14) applies.
- For a sequential composition $p; q$, when p does not terminate normally, the inductive hypothesis for the structurally smaller p concludes the proof. Otherwise, the interval $I^{\mathsf{X}} = I_1^{\mathsf{X}} \fatsemi I_2^{\mathsf{X}}$ splits into two parts that are nonempty by Proposition 4.3. For $k < \#I_1^{\mathsf{X}}$ the k-th step occurs in the smaller p. Otherwise, step k is done by q, which is covered by the hypothesis for $I^{\mathsf{X}\prime} := I_2^{\mathsf{X}}$ and $k' := k - \#I_1^{\mathsf{X}}$, using Lemma 4.23 (see below) to split off the I_1^{X} part of the execution.
- The cases omitted in Definition 4.21 just invoke an appropriate induction hypothesis for either a structurally smaller program or a shortened interval resp. smaller k. For **choose**, the second line of (4.12) ensures that the local states I^{X} of the submachine are unaffected by introduction of the local variables. □

The following lemma connects sequential composition of intervals to executions and runs. The lemma allows one to build larger executions from smaller fragments.

Lemma 4.23 (Composing executions)**.** *Sequential composition of intervals distributes through executions alongside labels*

$$I_1 \fatsemi \cdots \fatsemi I_n \in \mathit{execs}^T(\underline{j}_1..\underline{j}_n,\ \underline{i}_1..\underline{i}_n,\ \underline{o}_n..\underline{o}_1) \iff I_i \in \mathit{execs}^T(\underline{j}_i, \underline{i}_i, \underline{o}_i) \text{ for all } i \le n,$$

provided that the composition is defined, i.e., all I_i except for the last one are finite and $I_i.\mathit{last} = I_{i+1}.\mathit{first}$ for all $i < n$. If I_1 is a run starting in an initial state, then trivially I is one as well. Furthermore, an infinite concatenation when all the I_i are finite distributes similarly

$$I_1 \fatsemi I_2 \fatsemi \cdots \in \mathit{execs}^T(\underline{j}_1..,\ \underline{i}_1..,\ \underline{o}_1..) \iff I_i \in \mathit{execs}^T(\underline{j}_i, \underline{i}_i, \underline{o}_i) \text{ for all } i \le n.$$

Proof. In the binary case for $I_1 \fatsemi I_2$ by point wise analysis of its steps k. The key observation is that $k \ge \#I_1$ implies $(I_1 \fatsemi I_2)(k) = I_2(k - \#I_1)$; a similar property holds for the sequences of observables. The general case for a finite split follows by induction on n. For an infinite split, the $\Longrightarrow$ direction goes by induction on i from the right hand side, whereas if the left hand side is falsified in the $\Longleftarrow$ direction there is an I_i for the earliest step k in $I_1 \fatsemi I_2 \fatsemi \cdots$ that is not a transition contradicting the right-hand side. □

4.6 Related Work

This section discusses other approaches that relate to fine-grained program semantics as well as to the hierarchical specification of state-based systems. A detailed comparison to data refinement [77, 83], which considers data types with operations will be subject to Section 5.4.1

4.6.1 Trace Semantics

The idea to use traces (intervals) to model the execution of programs or transition systems is standard. The *Book of Traces* by Diekert and Rozenberg [43] gives a good introduction and overview.

Early work on trace semantics is for example by Brookes [22]. He considers a while language equivalent to Definition 3.2 where the semantics of programs $[\![p]\!]$ in his notation is a set of traces corresponding to $\{I \mid I \models p\}$ in the notation of the thesis. The rule system is set up just like Definition 4.2, although the least fixpoint is used and the semantics for loops is given explicitly as the union of all traces that are finite and end with a state where the loop's guard is false and those traces where the guard is infinitely often enabled. Brookes shows equivalence to a characterization with the greatest fixpoint that is similar to the definition for **`while`** in our work [55]. A technical achievement of Brookes [22] which should be mentioned for completeness of the discussion is a proof that the semantics is "fully abstract" with respect to an equivalent operational one, although the result will not be relevant in this thesis.

Semantics based on greatest fixpoints occur for example in [70] and [115]. The latter work presents and compares four different semantics for a while-language (relational and functional ones) each formulated a big step and a small step fashion. The development is fully mechanized in the Coq proof assistant [20], which has native support for coinductive definitions and proofs. Nakata and Uustalu [115] define two judgements

$(p, s) \Rightarrow I$, corresponding to $I.\mathit{first} = s$ and $I \models p$

$(p, I_1) \overset{*}{\Rightarrow} I$, corresponding to $I = I_1 \fatsemi I_2$ for some I_2 such that $I_2 \models p$ when I_1 is finite

This particular definition of sequential composition is fully constructive and thereby fits well within the mathematical framework of Coq (implying that one cannot rely on case distinctions whether an interval is finite or not). In the paper, the authors give some nice examples what happens in case of a non-productive rule system, for example that observations can be made "after" an infinite loop.

The logic presented by Bubel, Din, Hähnle, and Nakata [23] can express liveness and safety properties over the traces $I \models p$ of sequential programs. The focus in this paper is to define proof rules that can be applied in automated reasoning at the level of individual program steps, which is complementary to the work in this thesis.

Traces are typically used as models of concurrent systems, such as the failure & divergences semantics of Communicating Sequential Processes (CSP) [86]. Logics to reason about the temporal behavior include Linear Time Logic (LTL), Computational Tree Logic (CTL), or Interval Temporal Logic (ITL). Specific instance and terminology in this thesis inspired by the latter (ITL) [31, 110].

For the logic RGITL [145], a deep embedding into the KIV prover has been done in order to verify the deduction rules of symbolic execution for temporal logic programs.[2] A similar effort has been made by the author to study the fine-grained interval semantics as presented in Definition 4.2 and to validate the lemmas in Section 4.1, leading to some clarifications in the former approach related to the semantics of procedure calls.

A generalization is to replace the total sequential ordering of steps with a partial order, which leads to the model of partially ordered multiset (pomset) by Pratt [128]. This model permits to switch the degree of atomicity and abstraction somewhat similarly to

[2] https://swt.informatik.uni-augsburg.de/swt/projects/RGITL.html

the submachine call mechanism in this work, although in [128] no calling that exploits this feature is described, whereas this aspect is central here. Further exploring similarities to the pomset model is left as future work.

4.6.2 Reactive Systems

Event based systems such as CSP [86] communicate via shared events that synchronize the steps of systems running in parallel.

TLA$^+$ by Lamport [99] is a specification language designed to model concurrent and distributed systems. It is based on execution traces that are made "stutter-invariant", which introduces freedom to collapse the internal steps of a system. Communication is exclusively modeled by shared variables, instead of explicit observations as in this thesis.

Action systems [14] partition the state of a system into an internal part a and a global part u, where the latter is visible from the outside only. Transitions of the system are characterized by external actions nA_j for indices j

$$\exists\, j. nA_j(a_i, u_i)(a_{i+1}, u_{i+1})$$

and stuttering actions

$$nA_\sharp(a_i, u_i)(a_{i+1}, u_{i+1}) \implies u_i = u_{i+1}$$

which leave the global state is unchanged but the internal one may do something. Note that these are different from our transitions τ, which leave do not change the internal as well.

The behavioral semantics of action systems considers traces *beh* that contain the external actions only. The fundamental composition of systems is a parallel one that interleaves the steps of the system. A sequential calling protocol similar to our submachines is not considered [14, Section 3.3]. The interaction of parallel systems versus sequential submachine calls will be discussed for an ASM based approach in the next section.

4.6.3 Abstract State Machines

Abstract State Machines were originally introduced under the name "evolving algebras" by Gurevich [72] with the intention to model the steps of any sequential algorithm (ASM postulate). The fundamental idea was that the states of an ASM would correspond to mathematical structures, in other words algebras. The signature of logical functions is partitioned into a static part (including e.g. mathematical operators) and a dynamic part, which is updated by assignments.

In his thesis, Schellhorn [137] defines an encoding of ASMs into Dynamic Logic [74], where algebras $\mathcal{A}$ interpret the static signature only. Dynamic functions are instead mapped to (higher order) function variables or maps (cf. Section 3.1) which interpreted by valuations ν. This partitioning is taken here as well. The difference between the two approaches is marginal and is mostly reflected in terminology.

With respect to standard ASMs, our syntax only uses a fragment of the syntax available in [29]. In particular we use parallel updates only in the atomic updates, while control state ASMs allow arbitrary ASM rules.

For the semantics given in Definition 4.2 it is not difficult to show that it agrees with standard semantics of ASMs in the following sense: A finite interval $(s, \cdots, s') \models p$ corresponds to a successful computation $[\![p]\!](s) \triangleright U$ of a consistent set of updates U that is

applied to s to get the final state $s' = s \oplus U$. Likewise, the atomic semantics of operations $(i, s, s', o) \in [\![\mathtt{Op}]\!]$ corresponds to the computation of a Turbo ASM rule in [28] and [29, Chapter 4] such that $s' = \bot$ corresponds to either a diverging computation of updates, or to the computation of an inconsistent set.

It would probably be possible to generalize the atomic steps to general ASM rules and to admit synchronous parallelism (**par** and **forall** constructs). However, this would complicate code generation as well as symbolic execution, since parallel rules may have *clashes* (cf. the calculus by Stärk and Nanchen [149] and [29, Chapter 8]). Initial work to recognize these and to recover a simpler semantics without update sets and a corresponding calculus is for instance by Schellhorn et al. [146].

Integrating arbitrary (synchronous) parallelism by the ASM construct **par** would furthermore require an extended notion of runs to express the constraints when it is admitted to call interface operations in parallel, i.e., to express how the M(X) program

$$\mathtt{Op}^{\mathtt{X}}_{j_1}(\underline{e}_1; \underline{z}_1) \ \mathbf{par} \ \mathtt{Op}^{\mathtt{X}}_{j_2}(\underline{e}_2; \underline{z}_2)$$

is mapped to an execution of X in accordance with Lemma 4.22. Such an extension would deal problems similar to the specification of concurrent interfaces, which is fairly complex (see for example [38] for an overview).

The submachine concepts for ASMs described by Börger and Schmid [28] differ from the ones in this work: A submachine in [28] is simply a named procedure that may have local state. This state, however, is not preserved across several calls, and hence just a work-around for providing assignments to local variables (otherwise introduced by **let** and **choose**).

The standard approach to compose ASMs is via *external* functions, which are partitioned into inputs and outputs. Input (also called monitored) functions can be read but not written to by a machine—its values are provided by the environment afresh for each step. Output locations are conversely written by a machine for the environment to observe. The definition of "firing of updates" in [29] admits any new value for input locations after computation $(s_k, s') \in [\![p]\!]$ of the main program p. For each dynamic function f:

$$s_{k+1}(f) = \begin{cases} s'(f), & \text{if } f \text{ is controlled by the machine,} \\ \text{arbitrary}, & \text{when it is an input function.} \end{cases} \tag{4.15}$$

In order to prove something meaningful of an interactive machine, the environment must be specified by additional means. Labeling the transitions as in Definition 4.8 formally gives a handle on this.

Remark. The presentation in [55, Definition 3] did rely on resetting the inputs to arbitrary values to accommodate for environment actions, resulting in an unnatural formalization artifact. With the present Definition 4.8, copying the input parameters happens explicitly and right *before* an operation is run (not after as in the above update), which is more natural in our opinion.

Note also that when the input $\underline{i}(k+1)$ is taken from the parameter $\underline{i}$ of $runs(\underline{i}, \cdots)$ the nondeterministic choice in (4.15) is explicitly resolved, whereas the ASM formalism relies on syntactic choice functions that are typically left implicit due to the notational overhead they would introduce in a fully precise presentation.

Another approach to set the values of input functions from the execution of other machines is presented by Nicolosi Asmundo and Riccobene [117] as the so-called *component*

composition $M_1 \otimes \cdots \otimes M_n$. The machines M_i execute in parallel and inputs from one machine are provided by the outputs of another, while ordinary state is kept private (by an embedding into the distributed ASM framework). This setup resembles the synchronization of actions in process calculi such as CSP [86] and Action Systems [14].

A major concern in [117] is to express the conditions so that $M_1 \otimes \cdots \otimes M_n$ produces no (additional) clashes and that the signatures are compatible. From the perspective of this thesis, an interesting result is stated in [117, Section 7.1], which is the parallel equivalent of the sequential Lemma 4.22 (extracting runs of subcomponents):

> "$\otimes$**-Property.** Given a system $M = \otimes_{i=1}^{n} M_i$, for every component M_i of M, the set of runs of the projection of M_i over M is included in the set of runs of M_i."

This result is interpreted as behavioral correctness of the composed system. The paper proceeds (Section 7.2) to relate safety and liveness properties formulated over the values of external functions to the runs of the composition (demonstrated by an example). The formal connection how safety and liveness propagate through $\otimes$ is made in Nicolosi Asmundo's thesis [11, Section 6.5.3], essentially stating that projection of runs onto external functions distributes over $\otimes$ and that $\otimes$ is monotone. It is left open how $\otimes$ relates to ASM refinement.

Submachine composition `M(X)` can in principle be encoded as component composition `M` $\otimes$ `X` when calls to `X` are integrated via a protocol that synchronizes whether `M` or `X` executes and parameters to operations of `X` can be passed via external functions. Abstraction of many steps of `X` into one as done by the calling mechanism in Definition 4.14 can be modelled by compressing these into one Turbo step [28].

In this thesis, submachine calls are *first class* in the sense that they are built in—there is no need to encode them explicitly every time the concept of a call is used. The major hassle associated with encoding such calls is that it introduces additional control state in order to split up the steps of `M` at sequential compositions so that `M` can actually wait until the call returns. Börger and Schmid [28] argue in accordance with our experience that this often leads to undesired overhead in practice.

Besides supporting a more direct modeling approach where the required concepts of interfaces and calls can be expressed, our approach has the additional benefit that semantic properties can be proved on the meta level, notably submachine substitution (Theorem 5.9) and the extension to crashes (Chapter 11).

The recent thesis of Zenzaro [162] aims at "language level solutions" to address lacking modularity in the ASM formalism (see Section 3.3). The work is fundamentally tied to the CoreASM infrastructure [56]. While this is not a limitation on its own, the concepts defined in [162] are often just characterized by the code of the interpreter that executes them, referring to special locations such as *selectedRule*, which have no clear semantic manifestation. It is sometimes hard to determine their rigorous mathematical counterpart, although some concepts are described by inference rules. The real issues of modularity—a semantic account of compositionality—is bypassed completely, which makes Zenzaro's work valuable for modeling purposes but not for formal verification.

4.6.4 B and Event-B

The B-Method by Abrial [5] also called classical B and its simplified successor Event-B [6] emphasize structural aspects of systems and an incremental, refinement-based approach.

Classical B permits one to define machines that are similar to data type like machines and to employ different composition patterns as described in detail in [126]. The underlying logic is set theory, where "contexts" take the role of the algebraic specifications in this work (cf. Section 3.1). Besides submachine composition, "read-only" access to internals of other machines is permitted as well. The restriction that the state of machines is inaccessible from the outside is thereby relaxed somewhat. Potet and Rouzaud [126] give the conditions under which this is correct in the sense that refinement relations are preserved.

The simplifications over B introduced in Event B affect the internal structure of events, which correspond to the operations in this work: The programs used in events are restricted to parallel assignments—no sequential composition, loops, or calls are permitted. This significantly streamlines the theory in comparison to this work resp. the classical B (which supports these constructs in implementation level machines). The decision behind this shift is twofold: on one hand, Event B is intended for the modeling of systems instead of programs, on the other hand, a simpler theory supposedly leads to a more approachable method for non-experts. This claim is supported by recent success and applications of the Rodin Platform [7],[3] an Eclipse-based environment for Event B. However, the simplifications of Event-B make the formalism unsuitable for developments such as the Flashix file system, which benefits much from expressive programming language features.

[3] http://www.event-b.org

Chapter 5

Modular Refinement

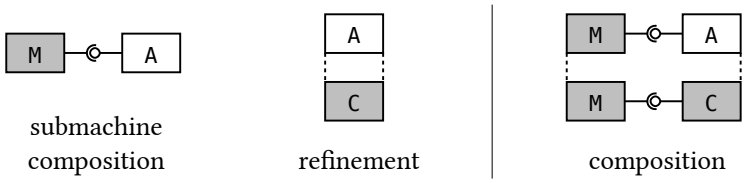
I'm not sure. I'm afraid we need to use . . . math!
— Hubert J. Farnsworth

Summary. This chapter regards the incremental development of hierarchical systems using *refinement*: An abstract specification of the system is gradually transformed into a concrete implementation with proofs supporting the preservation of behavior. The focus lies on nested machines and compositional replacement of submachines within the context of an outer supermachine. A novel technical result lifts refinement expressed in terms of runs over sequential submachine composition, independently of the method used to prove refinement.

Publications: This chapter is based on [52, 55].

Contents

In this chapter, the refinement method underlying an incremental system development is given. Specifically, refinement of an abstract machine A to a concrete machine C—written $\mathtt{A} \sqsubseteq \mathtt{C}$ and pronounced "A is refined by C"—expresses that C adheres to it specification A. Refinements are visualized by dashed lines as shown in the diagram below.



The main result of this chapter connects submachine composition and refinement:

Theorem (Compositionality). $\mathtt{A} \sqsubseteq \mathtt{C} \implies \mathtt{M}(\mathtt{A}) \sqsubseteq \mathtt{M}(\mathtt{C})$.

Graphically, this means that the two diagrams can be composed as shown on the right i.e., the grey parts can be linked together as a system $\mathtt{M}(\mathtt{C})$ representing the final code that at runs at the end. The theorem guarantees that this system is correct with respect to the compound specification $\mathtt{M}(\mathtt{A})$.

Compositionality of refinement used in this thesis is based entirely on the input/output behavior of machines (cf. Section 3.4): it will be shown that inclusion of the runs of `C` within the ones of `A` (modulo nontermination of operations) is a sufficient criterion for replacement within a *sequential* context. Surprisingly, in the literature such a theorem has only been proved for the strictly stronger conditions of forward and backward simulation: He, Hoare, and Sanders [77] as well as de Roever and Engelhardt [41, Section 4.4] prove it for data refinement, Gardiner and Morgan [64] give an account in the refinement calculus.

However, it may not always be appropriate to fix a particular proof method beforehand (either forward or backward simulation). For example, one future extension planned for Flashix is concurrency, where criteria such as *linearizability* [80] or *serializability* are primarily expressed in terms of runs and are not necessarily amenable to simulation proofs, because they reorder steps of a system. This means that the theorems in [41, 64] will not help. On the other hand, Filipović et al. [58] and Schellhorn et al. [143] have shown that linearizability is equivalent to observational refinement.

Therefore, this thesis will take the weaker prerequisite of inclusion of input/output behavior as foundation, which leads to a more general compositionality theorem that is independent of the proof method but nevertheless makes observations explicit.

In the ASM refinement method, which *is* based on runs, the notions of machine interfaces, contexts and observational behavior are intentionally not fixed [27, Section 3]. It is argued that the conditions given there for correct refinements are flexible to enough cover a wide range of applications and this claim is supported well by the examples given. However, the price to pay is that any compositional reasoning must be justified in an somewhat ad-hoc fashion. This works on paper but a mechanized approach clearly needs a more restrictive discipline to scale.

5.1 Trace Refinement

The idea behind refinement $A \sqsubseteq C$ of data type like machines for a specification A to an implementation C is simple: A client or context of machine A can only see the called operation, inputs, and outputs, whereas the internal state of A is hidden (cf. Definition 4.14 of the semantics of submachine calls). Therefore, replacing A by an implementation C that can engage in the same interactions (i.e., gives the same outputs for a particular input) does not affect the context in any way. Therefore, this refinement theory will be based on modeling observations explicitly (cf. Section 3.4). The formal definition is introduced at the level of transition systems:

Definition 5.1 (Refinement of transition system)**.** An abstract transition system $A = (ASt, Init^A, \longrightarrow_A)$, correctly refined to $C = (CSt, Init^C, \longrightarrow_C)$ written $A \sqsubseteq C$, iff for all I^C and sequences of observations $\underline{j}, \underline{i}, \underline{o}$

$$I^C \in runs^C(\underline{j}, \underline{i}, \underline{o}) \quad \Longrightarrow \quad \exists\, I^A \in runs^A(\underline{j}, \underline{i}, \underline{o}) \text{ such that } I^A \sqsubseteq I^C \tag{5.1}$$

where correspondence $I^A \sqsubseteq I^C$ of two intervals requires that their lengths is the same and that the concrete system diverges only after the abstract one permits it:

$$\begin{aligned} I^A \sqsubseteq I^C \quad \Longleftrightarrow \quad & \#I^A = \#I^C \text{ and for all } k \leq \#I^A \\ & I^C(k) = \bot \implies I^A(k) = \bot \end{aligned} \tag{5.2}$$

The requirements are discussed in turn.

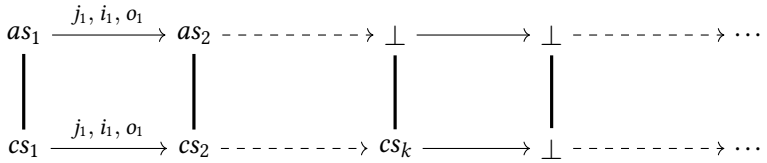


Figure 5.1: Refinement between abstract run $(as_1, \cdots)$ and corresponding concrete run $(cs_1, \cdots)$, where the abstract run diverges some time after as_2.

- Condition (5.1) states that the input/output behavior of C is covered by its specification A, regardless of the internal states: whenever C exhibits a sequence of observations $\underline{j}, \underline{i}, \underline{o}$ then A must be able to reproduce this sequence in some way witnessed by the existentially quantified run I^A.
- Condition (5.2) specifies that if C diverges then this must also be covered by the specification. According to Definition 4.8, there are two reasons when this is admitted by A due to $as \longrightarrow_A \bot$: The abstraction precondition is not satisfied, or alternatively, the abstraction operation does not terminatge. Both are considered as equally bad and impose no requirement on C. In Figure 5.1 this happens at step k.

Lemma 5.2 (Properties of refinement)**.** *Refinement is reflexive and transitive:*

$$A \sqsubseteq A \qquad \text{and} \qquad A \sqsubseteq B,\ B \sqsubseteq C \implies A \sqsubseteq C.$$

Proof. Obvious from Definitions 5.1. □

5.2 Forward simulation

Dealing with potentially infinite runs directly is cumbersome. It is more useful to rely on an inductive argument that breaks the conditions down to individual transitions.

Theorem 5.3 (Forward simulation)**.** *Refinement $A \sqsubseteq C$ can be proved by an inductive invariant $R \subseteq ASt \times CSt$ between states of A and states of C called a* forward simulation *such that*

$$Init^C \subseteq Init^A \fatsemi R \qquad \text{initialization}$$

$$R \fatsemi (\xrightarrow{j,i,o}_C) \subseteq (\xrightarrow{j,i,o}_A) \fatsemi R \quad \text{for all } j, i, o \qquad \text{correctness}$$

where $\fatsemi$ denotes relational composition. We postulate that $(\bot, cs) \in R$ for all $cs \in CSt$ to permit any cs when as is already $\bot$ in accordance to (5.2).

Purpose of the relation R is to accumulate additional information such as invariants of the two transition systems and a *coupling* between the two state spaces. The relation R expresses that the *internal* view of the states of A and C is consistent with each other. Forward simulation renders the components transparent in order to make incremental proofs possible.[1]

Figure 5.2 shows a commuting 1:1 diagram of two lockstep transitions from A and C for the "correctness" condition. For a given concrete transition $cs \xrightarrow{j,i,o} cs'$, a suitable abstract

[1] Backward simulation, which also modularizes refinement proofs, will not be considered here since it is applicable to finite runs only.

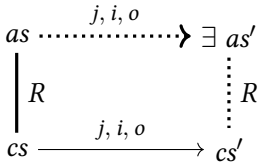


Figure 5.2: Forward simulation R with commuting 1:1 diagrams.

successor state as' has to be found that satisfies the fat, dashed lines: it is an A transition with observation j, i, o and R holds again. The ensuing proof for Theorem 5.3 concatenates such diagrams to construct a complete matching abstract run witnessing the refinement condition as graphically shown in Figure 5.2.

Proof of Theorem 5.3. From a run $I^C \in runs^C(\underline{j}, \underline{i}, \underline{o})$ its counterpart $I^A \in runs^A(\underline{j}, \underline{i}, \underline{o})$ of the same length is constructed so that $I^A \sqsubseteq I^C$ from (5.2) holds.

If I^C is finite then I^A is incrementally extended at the end by induction on the length of I^C. The base step invokes the initialization condition. The inductive step invokes the correctness condition for the last transition in I^C. In order to establish that R holds between the last states of the two intervals constructed so far as required by condition correctness, the inductive statement is strengthened so that we keep track of $R(I^A.last, I^C.last)$, which is the prerequisite of the correctness condition in the inductive step. In case $I^C.last = \bot$, however, $I^A.last = \bot$ canonically extends the trace.

Infinite runs require a diagonalization argument: for all k a corresponding interval $I^A_k \in runs^A(j(0), i(0), o(0), \cdots, j(k-1), i(k-1), o(k-1))$ of length k exists such that all shorter intervals from the construction are its prefixes, i.e.,

$$\begin{aligned} I^A_0 &= (as_0) \qquad \text{such that } as_0 \in Init^A \\ I^A_1 &= (as_0, as_1) \\ I^A_2 &= (as_0, as_1, as_2) \\ &\vdots \qquad\qquad\quad \ddots \end{aligned}$$

The abstract run $I^A = (as_0, as_1, as_2, \cdots)$ consists of the diagonal of last states such that $I^A(k) = I^A_k.last$—technically the construction invokes the axiom of choice to get a handle on all of these simultaneously. A stepwise analysis shows that I^A is indeed a run of A, because as_0 is initial and the respective last step of each I^A_{k+1}

$$as_k \xrightarrow{j(k),i(k),o(k)}_A as_{k+1}$$

is a valid transition as guaranteed by the fact that I^A_{k+1} is a run. The construction of the individual I^A_k mirrors the inductive argument from the first part of the proof for a finite I^C, although over the counter k instead of $\#I^C$. □

The syntactic proof obligations for data type like ASMs that imply refinement (of the corresponding transition systems) by Theorem 5.3 can be expressed in the weakest precondition calculus of Section 3.3.

Theorem 5.4 (Forward simulation of ASMs)**.** *Machine* $\mathtt{C} = (\underline{cx}\colon \mathtt{St}^{\mathtt{C}}, \mathtt{Init}^{\mathtt{C}}, \{\mathtt{Op}_j^{\mathtt{C}}\}_{j\in\mathcal{J}})$ *refines* $\mathtt{A} = (\underline{ax}\colon \mathtt{St}^{\mathtt{A}}, \mathtt{Init}^{\mathtt{A}}, \{\mathtt{Op}_j^{\mathtt{A}}\}_{j\in\mathcal{J}})$ *provided a forward simulation condition* $\mathtt{R}(\underline{ax}, \underline{cx})$ *such that*

$$\mathtt{Init}^{\mathtt{C}}(\underline{cx}) \vdash \exists\, \underline{ax}.\ \mathtt{Init}^{\mathtt{A}}(\underline{ax}) \wedge \mathtt{R}(\underline{ax}, \underline{cx}) \qquad \text{initialization}$$

$$\mathtt{R}(\underline{ax}, \underline{cx}), \mathtt{pre}_j^{\mathtt{A}}(in, \underline{ax}) \vdash \mathtt{pre}_j^{\mathtt{C}}(in, \underline{cx}) \qquad \text{applicability}$$

$$\mathtt{R}(\underline{ax}, \underline{cx}), \mathtt{pre}_j^{\mathtt{A}}(in, \underline{ax}), \langle\!|\mathtt{Op}_j^{\mathtt{A}}(in; \underline{ax}, out')|\!\rangle \mathtt{true} \qquad \text{correctness}$$
$$\vdash \langle\!|\mathtt{Op}_j^{\mathtt{C}}(in; \underline{cx}, out)|\!\rangle\ \langle \mathtt{Op}_j^{\mathtt{A}}(in; \underline{ax}, out')\rangle\ (out = out' \wedge \mathtt{R}(\underline{ax}, \underline{cx}))$$

The conditions reflect their semantic counterparts of Theorem 5.3. The first line guarantees the existence of matching initial states.

The second line propagates that the abstract precondition to the concrete one, ensuring that whenever some context calls the A machine, the C machine can also deal with the given inputs. The applicability condition is reflects its semantic counterpart (5.2) without directly referring to $\bot$-states that have no explicit representation in the algebraic specification of $\mathtt{St}^{\mathtt{A}}$ and $\mathtt{St}^{\mathtt{C}}$.

The third line expresses that when R holds, every execution ($\langle\!|\_|\!\rangle\_$) of the concrete ASM operation $\mathtt{Op}_j^{\mathtt{C}}$ leads to a state such that there is some corresponding abstract execution ($\langle\_\rangle\_$) of $\mathtt{Op}_j^{\mathtt{A}}$ with the same outputs that re-establishes R. Note that $\underline{ax}$ and $\underline{cx}$ in the postcondition of the modalities refers to modified states after executing both operations. Additionally, one may assume the (abstract) precondition. To prove the correctness condition, one may assume that the abstract system does something useful, i.e., precondition $\mathtt{pre}_j^{\mathtt{A}}$ holds and the abstract operation is guaranteed to terminate (written as $\langle\!|\mathtt{Op}_j^{\mathtt{A}}(in; \underline{ax}, out')|\!\rangle \mathtt{true}$).

Proof of Theorem 5.4. By Theorem 5.3. Initialization is trivial. The correctness condition of Theorem 5.3 is established by a case analysis of whether the abstract operation potentially diverges. If so, correctness holds trivially by the assumption that $(\bot, cs) \in R$. Otherwise, both the abstract precondition must hold as well as the abstract operation terminates, so that the third line applies. □

Schellhorn [137, 138] gives a general characterization of a variety of such refinement conditions. It is based on the original, more liberal definition of refinement for ASMs by Börger [27] and does therefore *not* establish the stronger notion of refinement according to Definition 5.1: The liberal definition works for machines only that can assume full control of their execution, which is not adequate for data type like ASMs.

Example 5.5 (Refinement of the Mini File System)**.** We continue the development of the idealized file system of Example 4.12 by a refinement that introduces out-of-place updates similar to its realistic counterpart in Flashix, the flash file system core, which is explained in Chapter 9.

Recall that the Mini FS provides four operations, namely `create`, `delete`, `read`, and `write`. Its internal state is a mapping $fs\colon \mathtt{Name} \rightharpoonup \mathtt{File}$ from names to file content.

This state is now broken down to an index $index\colon \mathtt{Name} \rightharpoonup \mathtt{Address}$ that tracks locations of files on a flash store $disk\colon \mathtt{Address} \rightharpoonup \mathtt{File}$. The indirection via addresses $adr \in \mathtt{Address}$ permits to move data around on the disk storage to fresh locations. The idea is that the abstract file store can be reconstructed by function composition $fs = disk \circ index$, i.e., lookup is now a two-stage operation $fs[name] = disk[index[name]]$. In order for this

composition to be well defined, we state the invariant $\texttt{ran}(index) \subseteq \texttt{dom}(disk)$ that must be maintained by operations.

The refined operations are shown below in Figure 5.3. Two aspects are new in comparison to Figure 4.2: The operation `create` now lazily postpones allocation of the file content on disk by recording a special address $\texttt{null} \in \texttt{Address}$ in the index. We establish that $disk[\texttt{null}] = \texttt{empty}$, i.e., there is only one incarnation of the empty file content as an optimization to save some disk space (we could choose not to store this file and adapt the `read` operation, but then the coupling $fs = disk \circ index$ would be more complicated).

The initial state is thus $disk$ with $disk[\texttt{null}] = \texttt{empty}$ and $index = \varnothing$.

The second difference to the Mini FS is the indirection via addresses. This manifests in the `write` operation, which chooses a fresh location adr for the new content $file$ to which the data is written. The index is updated alongside to refer to this new location, but only after the file has actually reached the disk.

This strategy gives a hint about power cut mitigation. Suppose a crash happens that interrupts the assignment $disk[adr] := file$, leaving partially written data at the address adr, then the index (provided it can be recovered) still refers to the previous content. As long as the $disk$ is not cleaned up by removing obsolete entries $adr \notin \texttt{ran}(index)$, this system is power cut safe.

```
create(name)
  precondition ¬ name ∈ dom(index)
  index[name] := null

delete(name)
  precondition name ∈ dom(index)
  index := index − name

read(name; file)
  precondition name ∈ dom(index)
  file := disk[index[name]]

write(name, file)
  precondition name ∈ dom(index)
  choose adr ∉ dom(disk) in
    disk[adr] := file;
    index[name] := adr
```

Figure 5.3: Refinement of Mini FS with flash specific out-of-place updates.

We briefly demonstrate how to verify the refinement by forward simulation. The coupling matches the two states and contains the invariants:

$$\texttt{R}(fs, index, disk) \;:=\; fs = disk \circ index \wedge \texttt{ran}(index) \subseteq \texttt{dom}(disk) \wedge disk[\texttt{null}] = \texttt{empty}$$

- Initialization: $\texttt{R}(\varnothing, \varnothing, disk)$ holds trivially.
- Applicability: From the composition $disk \circ index$ and the invariant about the range of the index from `R` we have that $\texttt{dom}(index) = \texttt{dom}(fs)$.
- Correctness, shown for `write`. We have to prove

  $$\texttt{R}(fs, index, disk), name \in fs$$
  $$\vdash \langle\!|\texttt{write}^{\texttt{C}}(name, file)|\!\rangle \; \langle \texttt{write}^{\texttt{A}}(name, file) \rangle \; \texttt{R}(fs, index, disk).$$

  By symbolic execution of the modalities, the remaining predicate logic goal is

  $$\texttt{R}(fs, index, disk), name \in fs, adr \notin \texttt{dom}(disk)$$
  $$\vdash \texttt{R}(fs[name \mapsto file], index[name \mapsto adr], disk[adr \mapsto file])$$

  where adr is the fresh address chosen by the concrete `write` operation (cf. Figure 5.3). Checking that this condition holds is trivial. ■

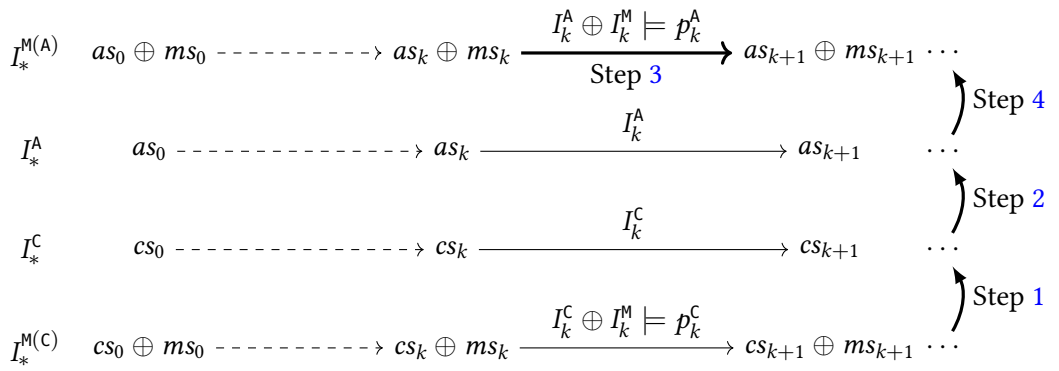


Figure 5.4: Substitution of a concrete run $I^{\mathtt{C}}$ by an abstract one $I^{\mathtt{A}}$ within the context of a machine $\mathtt{M}$. The k-th call to an operation of $\mathtt{M}$ is singled out. The bold arrow at the top represents substitution of $\mathtt{C}$ for $\mathtt{A}$ in the operation's program, which is central to the proof.

5.3 Submachine Refinement

Syntactically, substituting (or "linking") a given machine $\mathtt{A}$ for $\mathtt{X}$ in $\mathtt{M}(\mathtt{X})$ produces the compound system $\mathtt{M}(\mathtt{A})$, which is defined by structurally replacing calls to $\mathtt{Op}_j^{\mathtt{X}}$ by their counterparts $\mathtt{Op}_j^{\mathtt{A}}$ in the programs defining the operations of the context $\mathtt{M}$. The replacement at the level of programs is written as $p\{\mathtt{X} \mapsto \mathtt{A}\}$. In the following it is assumed that $\mathtt{X}$ and $\mathtt{A}$ are syntactically compatible in the sense that they are defined over the same set of indices $\mathcal{J}$ and that the signatures of operations are the same.

Purpose of this section is to formally prove that refinement $\mathtt{A} \sqsubseteq \mathtt{C}$ between an abstract machine $\mathtt{A}$ and a concrete machine $\mathtt{C}$ propagates to the context, i.e., $\mathtt{M}(\mathtt{A}) \sqsubseteq \mathtt{M}(\mathtt{C})$ holds.

Although forward simulation is incomplete in general (see e.g. [41, Section 2.2.2]) it is nevertheless stronger than refinement as it guarantees that the same run can be extended by a step at the end, whereas refinement may pick any I^A for a given I^C, in particular, abstract runs of different lengths may be entirely unrelated. This aspect must be taken into account to prove the desired compositionality: one must show that a fixed but arbitrary concrete run $I_*^{\mathtt{M}(\mathtt{C})}$ maps to some abstract run $I_*^{\mathtt{M}(\mathtt{A})}$ *as a whole*–it will not be possible to construct this $I_*^{\mathtt{M}(\mathtt{A})}$ incrementally.

Proof Outline.

The high-level argument can be followed graphically in Figure 5.4 along the textual description below. The argument proceeds from the given, concrete interval $(cs_0 \oplus ms_0, \dots)$ at the bottom to construct the corresponding abstract interval $(as_0 \oplus ms_0, \dots)$ at the top.

Step 1 (Extraction)**.** From each transition $I_*^{\mathtt{M}(\mathtt{C})}(k) \longrightarrow I_*^{\mathtt{M}(\mathtt{C})}(k+1)$ of the global run shown at the bottom in Figure 5.4 of the concrete compound system an execution fragment $I_k^{\mathtt{C}}$ is extracted that captures the steps of the program of the k-th call of an $\mathtt{M}$ operation by Lemma 4.22. These can be composed to a stuttering run $I_*^{\mathtt{C}} = I_1^{\mathtt{C}} \fatsemi I_2^{\mathtt{C}} \fatsemi \cdots$ of $\mathtt{C}$. Similarly, the remainder of the state for $\mathtt{M}$ is collected in an interval $I^{\mathtt{M}} = I_1^{\mathtt{M}} \fatsemi I_2^{\mathtt{M}} \fatsemi \cdots$ with splits of the same lengths. The depicted states as_k, cs_k, and ms_k correspond to the ones shared by the

respective adjacent intervals with $I_*^{\mathtt{M}(\mathtt{C})}(k) = (cs_k \oplus ms_k(in \mapsto i))$ (for some inputs i of $\mathtt{M}$) and $cs_k = I_k^{\mathtt{C}}.first = I_{k-1}^{\mathtt{C}}.last$ (likewise for ms_k).

Step 2 (Mapping)**.** Refinement $\mathtt{A} \sqsubseteq \mathtt{C}$ guarantees the existence of a matching abstract run $I_*^{\mathtt{A}} = I_1^{\mathtt{A}} \fatsemi I_2^{\mathtt{A}} \fatsemi \cdots$ consisting of execution fragments of $\mathtt{A}$ with same lengths as the corresponding $I_k^{\mathtt{C}}$. The critical aspect of this step is that the run of $\mathtt{C}$ must be mapped as a whole. The mapping is depicted as the two runs in the middle in Figure 5.4.

Step 3 (Substitution)**.** The central technical part of the proof is to show that it is irrelevant from the perspective of $\mathtt{M}$ which submachine it calls. For any two matching intervals $I^{\mathtt{A}} \sqsubseteq I^{\mathtt{C}}$, $I^{\mathtt{C}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathtt{C}}$ implies $I^{\mathtt{A}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathtt{A}}$, i.e., $I^{\mathtt{A}}$ is compatible with $I^{\mathtt{M}}$ when $I^{\mathtt{C}}$ is (for the same calls $\underline{j}$, inputs $\underline{i}$, and outputs $\underline{o}$ of the two submachines). Here, $p^{\mathtt{C}}$ is any program calling operations of $\mathtt{C}$ and $p^{\mathtt{A}} := p^{\mathtt{C}}\{\mathtt{C} \mapsto \mathtt{A}\}$ is its counterpart where calls have been redirected to $\mathtt{A}$.

Step 4 (Recombination)**.** It remains to be shown that the complete abstract run $I_*^{\mathtt{A}}$ constructed in this manner fits within the context by considering each k-th operation of the global $\mathtt{M}$ run individually. In the figure, this subpart of the intervals is singled out, and the desired conclusion is represented by the bold arrow at the top, which is established by substitution (Step 3) for the instances $I^{\{\mathtt{A},\mathtt{C},\mathtt{M}\}} := I_k^{\{\mathtt{A},\mathtt{C},\mathtt{M}\}}$ and $p^{\{\mathtt{A},\mathtt{C}\}} := p_k^{\{\mathtt{A},\mathtt{C}\}}$.

To summarize, compositionality is lifted from to the level of the steps of a program and to the steps of the whole machine.

Step 1: Extraction

The extraction of $I^{\mathtt{C}}$ in Step 1 of the high-level argument stems from a characterization of the $\mathtt{M}(\mathtt{C})$ step $s \xrightarrow{j,i,o} s'$ that exposes this $I^{\mathtt{C}}$ explicitly by inlining the corresponding call $(i, s, s', o) \in [\![\mathtt{Op}_j^{\mathtt{M}}]\!]$ with the atomic semantics of operations Definition 4.8 and programs Definition 4.7. There are intervals $I^{\mathtt{C}}$, $I^{\mathtt{M}}$, a call sequence $\underline{j}$ as well as inputs $\underline{i}$/outputs $\underline{o}$ that capture the execution of the submachine throughout the program p_j belonging to $\mathtt{Op}_j^{\mathtt{M}}$, i.e.,

$$s \xrightarrow{j,i,o} s' \iff \exists\, I^{\mathtt{C}}, I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o}.\ \begin{cases} I^{\mathtt{C}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p_j, & \text{if } s \neq \bot \text{ and } s(in \mapsto i) \models \mathtt{pre}_j \\ I^{\mathtt{C}} = I^{\mathtt{M}} = (\bot), & \text{otherwise} \end{cases} \tag{5.3}$$

where $(I^{\mathtt{C}} \oplus I^{\mathtt{M}}).first = s(in \mapsto i)$ initially copies the input of the supermachine to the state, $s' := (I^{\mathtt{C}} \oplus I^{\mathtt{M}}).last$ is the last state of the combined interval, which also produces the output $o := s'(out)$. If this interval is infinite then $s' = \bot$.

By Lemma 4.22, $I^{\mathtt{C}} \in execs_\tau^{\mathtt{C}}(\underline{j}, \underline{i}, \underline{o})$, where the outcase in (5.3) produces a default one-state interval $(\bot)$ that composes nicely with subsequent steps (the choice is somewhat arbitrary but convenient).

The $I_k^{\mathtt{C}}$ provided by (5.3) at each step k in the global interval can be concatenated to the complete run $I_*^{\mathtt{C}}$ of $\mathtt{C}$. There are two cases

1. All the $I_k^{\mathtt{C}}$ are finite, then the potentially infinite $I_*^{\mathtt{C}} = I_1^{\mathtt{C}} \fatsemi I_2^{\mathtt{C}} \fatsemi \cdots$ is well defined (second construction in Lemma 4.23) because all overlapping pairs $I_k^{\mathtt{C}}.last = I_{k+1}^{\mathtt{C}}.first$ match by the above construction. Note that there is at most one $\bot$ state at the end of $I_*^{\mathtt{C}}$, because several one-state intervals $(\bot)$ are collapsed.

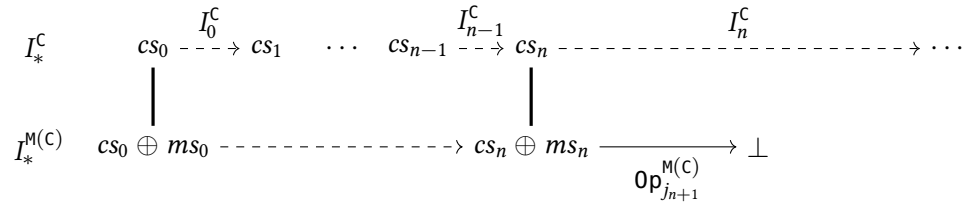


Figure 5.5: Divergence of operation $\mathtt{Op}^{\mathsf{M(C)}}$ at step n in the global interval $I^{\mathsf{M(C)}}_*$ shown at the bottom, inducing an infinite I^{C}_n. The reason for this can be either an infinite loop within the context itself, or divergence of a submachine call $\mathtt{Op}^{\mathsf{C}}_{j_{n+1}}$.

2. There is some infinite I^{C}_n, in which case $I^{\mathsf{C}}_* = I^{\mathsf{C}}_1 \fatsemi \cdots \fatsemi I^{\mathsf{C}}_n$ is well defined (first construction in Lemma 4.23) and cancels all $I^{\mathsf{C}}_{>n}$ in the sequential concatenation. All the states in the global interval $I^{\mathsf{M(C)}}_*$ from $n+1$ onwards are $\bot$, whereas the states up to n are proper ones since infinite I^{C}_n are only generated by program executions started with states different from $\bot$ in the fine grained semantics. The situation is visualized in Figure 5.5.

In both cases, the observables $\underline{j}_k$, $\underline{i}_k$, $\underline{o}_k$ are concatenated likewise to $\underline{j}_*$, $\underline{i}_*$, $\underline{o}_*$ and the composed interval I^{C}_* is a stuttering execution of C by Lemma 4.23. From the initialization of the compound system $\mathsf{M}(\mathsf{C})$ (Definition 4.13) and the initial state of the global run we can conclude that it is in fact a stuttering run $I^{\mathsf{C}}_* \in runs^{\mathsf{C}}_\tau(\underline{j}_*, \underline{i}_*, \underline{o}_*)$.

Step 2: Mapping

Proceeding with the high-level proof Step 2, refinement $\mathsf{A} \sqsubseteq \mathsf{C}$ maps the concrete run I^{C}_* to an abstract one $I^{\mathsf{A}}_* \in runs^{\mathsf{A}}_\tau(\underline{j}_*, \underline{i}_*, \underline{o}_*)$ for the same observations. Fortunately, this aspect is trivial by Definition 5.1. Since $I^{\mathsf{A}}_* \sqsubseteq I^{\mathsf{C}}_*$, the two intervals must have the same length and it is possible to replicate the shape of the concrete split as $I^{\mathsf{A}}_* = I^{\mathsf{A}}_1 \fatsemi I^{\mathsf{A}}_2 \fatsemi \cdots$.

Step 3: Substitution

Turning to the key step in the proof, with the enriched interval semantics substitution of calls at the level of programs becomes possible.

Lemma 5.6 (Substitution of submachine calls)**.** *For all executions $I^{\mathsf{C}} \in execs^{\mathsf{C}}_\tau(\underline{j}, \underline{i}, \underline{o})$ and executions $I^{\mathsf{A}} \in execs^{\mathsf{A}}_\tau(\underline{j}, \underline{i}, \underline{o})$ such that $I^{\mathsf{A}} \sqsubseteq I^{\mathsf{C}}$:*

$$I^{\mathsf{C}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathsf{C}} \quad \Longrightarrow \quad I^{\mathsf{A}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathsf{A}}.$$

Proof. By coinduction over the derivation of the consequence $I^{\mathsf{A}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathsf{A}}$ of the lemma. Dually to induction, it must be shown that the substitution property passes on from the conclusion to the premises of the respective rule deriving the interval for p^{A}. When there are no premises, the implication must be shown directly. In the following, "assumptions from the lemma" refers to the left hand side of the implication plus the condition about the respective executions. The interesting cases are sequential composition and submachine calls. The other cases affect just I^{M} in the same way on both sides of the implication. Note that $\#I^{\mathsf{C}} = \#I^{\mathsf{A}}$ because the length of the intervals coincides with $|\underline{j}|$ from the executions.

- Case $p \equiv \underline{x} := \underline{e}$. Assignments induce a stuttering step $\underline{j} = (\tau)$ so that from the executions, $I^{\mathsf{C}} = (cs, cs)$ and $I^{\mathsf{A}} = (as, as)$. This matches the semantics of assignments by (4.12) for the same I^{M} on both sides of the implication.
- Case $p^{\mathsf{C}} \equiv \mathtt{Op}^{\mathsf{C}}_j(\underline{e}; \underline{cx}, \underline{z})$ and $p^{\mathsf{A}} \equiv \mathtt{Op}^{\mathsf{A}}_j(\underline{e}; \underline{ax}, \underline{z})$. The non-diverging case for calls in Definition 4.21 provides $I^{\mathsf{A}} = (as, as')$ such that $as' \neq \bot$ and $\underline{j} = (j)$, $\underline{i} = (i)$, $\underline{o} = (o)$. It must be shown that the side conditions of the rule hold. By the definition of $I^{\mathsf{C}} \models \mathtt{Op}^{\mathsf{C}}_j(\underline{e}; \underline{cx}, \underline{z})$ we get the following facts: I^{C} is a one-step interval (cs, cs') such that $cs' \neq \bot$ from $I^{\mathsf{A}} \sqsubseteq I^{\mathsf{C}}$, and $(i, cs, cs', o) \in [\![\mathtt{Op}^{\mathsf{C}}_j(\underline{e}; \underline{cx}, \underline{z})]\!]$ holds. Since I^{A} is an execution for the given call sequence and inputs/outputs, $(i, as, as', o) \in [\![\mathtt{Op}^{\mathsf{A}}_j(\underline{e}; \underline{ax}, \underline{z})]\!]$ holds, regardless of whether $as' = \bot$. The other part of the side condition that $I^{\mathsf{M}} = (ms, ms')$ also matches the `A` call, because ms' depends on the output o only, which is the same (resp. can be in the case of $as' = \bot$) on both sides of the implication.
- Case $p \equiv p_1; p_2$.

  The case when p_1^{C} does not terminate just cancels the remaining program p_2 leading directly to the hypothesis (rule (4.2)).

  If p_1^{C} terminates, then the left hand side of $\Longrightarrow$ produces splits of the `C` and `M` intervals and the call sequence $\underline{j}$, i.e., $I^{\mathsf{M}} = I_1^{\mathsf{M}} \fatsemi I_2^{\mathsf{M}}$, $I^{\mathsf{C}} = I_1^{\mathsf{C}} \fatsemi I_2^{\mathsf{C}}$, $\underline{j} = \underline{j}_1\underline{j}_2$, $\underline{i} = \underline{i}_1\underline{i}_2$, and $\underline{o} = \underline{o}_1\underline{o}_2$. We fix $I_1^{\mathsf{A}} \fatsemi I_2^{\mathsf{A}} := I^{\mathsf{A}}$ such that $\#I_1^{\mathsf{A}} = \#I_1^{\mathsf{C}}$. Note that that the I_i^{A} for $i = 1, 2$ are finite stuttering `A` execution for $\underline{j}_i, \underline{i}_i, \underline{o}_i$ (by Lemma 4.23, and similarly for `C`).

  If $I^{\mathsf{A}}.last \neq \bot$ then the derivation

$$\frac{I_1^{\mathsf{A}} \oplus I_1^{\mathsf{M}}, \underline{j}_1, \underline{i}_1, \underline{o}_1 \models p_1^{\mathsf{A}} \qquad I_2^{\mathsf{A}} \oplus I_2^{\mathsf{M}}, \underline{j}_2, \underline{i}_2, \underline{o}_2 \models p_2^{\mathsf{A}}}{I^{\mathsf{A}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models (p_1; p_2)^{\mathsf{A}}}$$

  by rule (4.3) is valid and the premises are covered by the coinductive hypothesis.

  However, it is possible that a submachine call in p^{A} diverges on I_1^{A} already even if the concrete machine continues to function properly (cf. Definition 5.1). Then $I^{\mathsf{A}}.last = \bot$ and therefore $I^{\mathsf{A}}{\uparrow}$ holds and rule (4.2) applies:

$$\frac{I^{\mathsf{A}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p_1^{\mathsf{A}}}{I^{\mathsf{A}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models (p_1; p_2)^{\mathsf{A}}}$$

  However, because p_1^{C} terminates normally, the coinductive hypothesis just provides $I_1^{\mathsf{A}} \oplus I_1^{\mathsf{M}}, \underline{j}_1, \underline{i}_1, \underline{o}_1 \models p_1^{\mathsf{A}}$ which at least has the right program. Observe that the remainder $I_2^{\mathsf{A}} = (\bot, \cdots)$ of the split consists of $\bot$ states only, because $I_1^{\mathsf{A}}.last = \bot$ (similarly for I_2^{M}). The insight is that the interval semantics is robust against extension of such states at the end, which of course depends on the liberal Definition 4.14 of diverging submachine call. This is codified by Lemma 5.8 that closes the remaining gap.

The proof is concluded by a remark on the **`while`**-case: nothing essential happens when applying the rule, except that the program is exchanged (cf. Definition 4.2). This is perfectly fine as the argument does not depend on the (size) of the program in any way—unfolding the derivation of $I^{\mathsf{C}} \oplus I^{\mathsf{M}}, \underline{j}, \underline{i}, \underline{o} \models p^{\mathsf{C}}$ once by the rule for $p^{\mathsf{C}} = \mathbf{while}\ \varphi\ \mathbf{do}\ q^{\mathsf{C}}$ suffices to demonstrate that the assumptions from the lemma hold for the premise as well. □

Example 5.7 (Lack of expressiveness of the unannotated semantics $I \models p$.)**.** We briefly demonstrate where the proof would break down when the calls, inputs, and outputs are

not annotated to the program execution in the case for submachine calls, as argued in Example 4.20. We know that the intervals $I^{\mathsf{C}} \in \mathit{execs}^{\mathsf{C}}_{\tau}(\underline{j}, \underline{i}, \underline{o})$ and $I^{\mathsf{A}} \in \mathit{execs}^{\mathsf{A}}_{\tau}(\underline{j}, \underline{i}, \underline{o})$ match $I^{\mathsf{A}} \sqsubseteq I^{\mathsf{C}}$ for the specific call sequence $\underline{j}$ extracted by Lemma 4.22. However, for $I^{\mathsf{C}} \oplus I^{\mathsf{M}} \models p^{\mathsf{C}}$ from the assumptions, the case of a submachine call will give $p \equiv \mathtt{Op}_{j'}$ for a potentially *different* index $j' \neq j$. It is clearly not possible to conclude that I^{A} will model this call for the abstract submachine. The effect is prevented when the parameter $\underline{j}$ from the executions is linked to the program, which fixes $j' = j$ from the extended semantics of submachine calls in Definition 4.21.

Remark. The proof for the substitution lemma in [52] proceeds by induction on the structure of the program p and has a special case for **`while`** loops that explicitly iterates the construction. Relying on coinduction here unifies the argument and renders the cases for **`while`** and procedure calls obvious.

Lemma 5.8 ($\bot$-extension)**.** *For a finite I with $I.last = \bot$*

$$I \models p \quad \Longrightarrow \quad I \fatsemi \bot^{\omega} \models p.$$

For the extended semantics Definition 4.21 arbitrary sequences of calls, inputs and outputs can be appended analogously (not shown for brevity).

Proof. By induction on the finite derivation of $I \models p$ (Lemma 4.4), comparing the inference rules.[2] The extension is witnessed at some diverging submachine call that necessarily occurs from the fact that I already ends with $\bot$. □

Step 4: Recombination

Finally, the main result:

Theorem 5.9 (Compositionality)**.** $\mathtt{A} \sqsubseteq \mathtt{C} \implies \mathtt{M}(\mathtt{A}) \sqsubseteq \mathtt{M}(\mathtt{C})$.

Proof. It has already been demonstrated how to obtain $I^{\mathsf{C}}_{*} = I^{\mathsf{C}}_{1} \fatsemi I^{\mathsf{C}}_{2} \fatsemi \cdots$ and $I^{\mathsf{A}}_{*} = I^{\mathsf{A}}_{1} \fatsemi I^{\mathsf{A}}_{2} \fatsemi \cdots$ from the global run $I^{\mathsf{M}(\mathsf{C})}_{*} = I^{\mathsf{C}}_{*} \oplus I^{\mathsf{M}}_{*}$ of $\mathtt{M}(\mathtt{C})$. It remains to show that the interval composed of all the $I^{\mathsf{A}}_{k}.\mathit{first}$ and possibly some continuation with $\bot$-states is a run of $\mathtt{M}(\mathtt{A})$. Analogously to Step 1 there are again two cases:

- When all the intervals are finite, the concrete and the abstract operations match at each step k.
- The split is finite and all but the last intervals I^{C}_{n} and I^{A}_{n} are finite. The calls up to n match, and the global intervals both continue as $(\bot, \cdots)$ onwards.

Both cases follow from characterization (5.3) for the transition relation of $\mathtt{M}(\mathtt{C})$ resp. $\mathtt{M}(\mathtt{A})$. The inputs and outputs communicated between the operations of $\mathtt{M}(\_)$ to the outside world are reflected in the I^{M}_{k} only and are the same for the abstract and concrete runs. □

Example 5.10 (Compositionality for the Mini FS client)**.** Theorem 5.9 can be applied to the client of the toy file system described in Example 4.15. Recall that it had one operation, `copy`(*from*, *to*) to copy a file given by *from* to the new name *to*, which was implemented in terms of three submachine operations `read`, `create`, and `write` and a subsequent validation that read back the destination *to*.

[2] Coinduction on $I \fatsemi \bot^{\omega} \models p$ is also appropriate.

Using the flash specific implementation of Example 5.5 instead of the simple specification does not invalidate the assertion made: since the `copy` using the abstract submachine is guaranteed to terminate so is the `copy` refering to the concrete flash specific implementation (cf. Definition 5.1, specifically (5.2), for a one step run).

5.4 Related Work

This section picks up Section 3.4 and compares the approach of this thesis to different techniques.

5.4.1 Data Refinement

Data refinement [41, 42, 77, 83] is based on data types

$$DT = (St, Init, \{Op_j\}_{j\in\mathcal{J}}, Fin),$$

where each operation $Op_j \subseteq St \times St$ is a relation on states. Refinement considers programs $p(DT)$ that initialize the data type using $Init \subseteq GSt \times St$, call some operations, and then shut down the data type using $Fin \subseteq St \times GSt$, where GSt is the global state space of the context. It follows that programs $p(DT)$ using such a data type induce a relation between such global states, i.e., $[\![p(DT)]\!] \in GSt \times GSt$.

The global state provides what can be observed from the outside, whereas the intermediate state space St includes the representation of the data type that is not visible when considering a complete program (thanks to initialization and finalization). This permits to exchange the data type by a refined version without affecting the overall behavior of the program:

Definition (Data refinement)**.**

$$ADT \sqsubseteq CDT \quad \Longleftrightarrow \quad \forall\, p.\ [\![p(CDT)]\!] \subseteq [\![p(ADT)]\!]$$

However, the operations of data types are just relations between states. There is no predefined calling mechanism that passes inputs and outputs back and forth. In [41, Section 2.3.2], its argued that the context program (of a supermachine in this work) is composed of such operations *only*, collapsing the notion of steps done by the context with those that are "true" calls in this work. Instead, there is a distinction into normal variables (belonging to the context) and representation variables (submachines state) which are treated differently. In effect, this specializes St to be of the form $GSt \times LSt$.

To record the relevant observations the data type is typically enriched to store these as part of the state space so that this information can be extracted into the global state GSt at the end. In particular, one can store the complete history corresponding to the sequence of labels $\underline{j}, \underline{i}, \underline{o}$. The benefit is that the $Op_j \subseteq St \times St$ just maps each state to its potential successors and does not refer to inputs and outputs, simplifying the presentation (in fact, [77] is charmingly concise). See [159] for a canonical definition of such a finalization operation and [139] for a formalization into the ASM approach.

We have opted here to keep the the inputs and outputs of operations explicit. One reason is that it makes explicit what is taken as the semantics of machines without the need to define a finalization. More importantly, we can reason about partial or even infinite submachine executions, whereas data refinement works for finite executions only as at the end finalization must be applied.

To see that we do need infinite traces to correctly replace submachine behavior, consider the simple program of a composed system $\mathtt{M}(\mathtt{C})$, which calls the submachine operation $\mathtt{Op}^{\mathtt{C}}$ to determine whether the loop should be exited.

$$\textbf{while } b \textbf{ do } \mathtt{Op}^{\mathtt{C}}(; b)$$

For an infinite execution where b is always true and a refinement $\mathtt{A} \sqsubseteq \mathtt{C}$, we need to be able to record and substitute the corresponding run of the submachine in order to justify that such a behavior is admitted by $\mathtt{A}$.

The forward simulation condition (called "downwards" in [77]) for a coupling relation $R \subseteq ASt \times CSt$ are analogous to the ones in Theorem 5.3:

$$\begin{aligned} Init^C &\subseteq Init^A \fatsemi R && \text{initialization} \\ R \fatsemi Op_j^C &\subseteq Op_j^A \fatsemi R \quad \text{for all } j \in \mathcal{J} && \text{correctness} \\ R \fatsemi Fin^C &\subseteq Fin^A && \text{finalization} \end{aligned}$$

Each concrete step Op^C must be witnessed by an abstract one Op^A. Note that no observations are made along the way: these are postponed on purpose to the finalization operation.

Swapping the sides of the relational compositions $\_ \fatsemi \_$ in the proof obligations dually leads to backward simulation. Together, forward and backward simulation provide complete proof method [77], i.e., Definition 5.4.1 can always be demonstrated by such a combination. However, backwards simulation works its way from the end of a run to the start, and it therefore not adequate without restrictions (like image finiteness in [103]) for infinite runs.

The embedding of nontermination is based on the contract-based approach of Z [159]. It can be viewed as an adaption of this approach to the setting of ASMs. We prefer the operational style of ASM rules over the relational style of Z operations, since ASMs can be executed and directly translated to code. Nevertheless, our atomic semantics (Definition 4.8) of ASM operations parallels the contract embedding of Z relations into states with bottom, except that we do not add $\{\bot\} \times S_\bot$, but just $\{\bot\} \times \{\bot\}$ to preserve the meaning of $\bot$ as "nontermination" (not "unspecified"). [139] argues that for both embeddings the same refinements are correct, in particular our simulation proof obligations are those of Z refinement.

The embedding when preconditions are violated is more strict in this work: Definition 4.8 in this thesis contains pairs $(s, \bot) \in [\![\mathtt{Op}]\!]$ when $s \not\models \mathtt{pre}$, whereas the standard approach is to permit *arbitrary* successor states $s' \in S \uplus \{\bot\}$. For backwards simulation the more liberal definition is indeed needed, because an abstract successor state as' is not subject to choice but given by the proof obligation for the respective commuting diagram. Specifically, one has to find as such that $(as, as') \in [\![\mathtt{Op}^{\mathtt{A}}]\!]$, and in the case when the precondition must not hold ($as \not\models \mathtt{pre}^{\mathtt{A}}$) we might have $as' \neq \bot$.

A similar approach is due to Gardiner and Morgan [64] in the context of refinement calculus: By using predicate transformers one can express a backwards rule (called "cosimulation" in this paper) that is complete on its own. However, as with data refinement in general, only finite traces are considered (the limitations are outlined below).

Divakaran et al. [47] consider similar modularity for data types, however, for finite runs only, and some restrictions on nondeterminism and coupling apply.

5.4.2 ASM Refinement

In the ASM approach, observations like j, i, o are part of the state space. The abstract and concrete states are connected by a relation $IO \subseteq ASt \times CSt$. This permits a high degree of freedom to express many different refinement concepts (see [27]) but makes it hard to have built-in modularity guarantees. In [55], which does not make the sequences $\underline{j}$, $\underline{i}$, $\underline{o}$ explicit, we have therefore restricted the relation IO for submachines to identity between input and output locations (cf. Section 5.2, equation (1) in [55]).

The definition given here is on the one hand more liberal than the one in [141], as it allows one to implement a diverging operation on the abstract level with any run on the concrete level. On the other hand it is more strict, as it forbids general m:n diagrams where $m > 1$ abstract operations are implemented with n concrete ones, since the environment cannot be forced to call a specific sequence of m operations.

5.4.3 Guards versus Preconditions

A major technical difference between our formalism and Event B [6] is that events can have guards only but not precondition (classical B has both), emphasizing that machines in Event B are systems interacting with an outside world instead of passive components being called by a client.

Guards are appropriate for *reactive* systems (see also [14]), where the system has full control over its internal flow of control and interacts via messages or events with the outside world, or alternatively using shared locations as in [99]. Here, calling an operation *forces* the submachine to react, i.e., whether an operation is executed is subject to the caller.

5.5 Discussion and Outlook

One can draw an analogy of "placeholder" machines `X` to regular variables x of predicate logic by introducing *machine variables* X that are interpreted by a corresponding machine environment μ, which takes the role of the valuation. The semantics of machines $[\![\mathtt{M}]\!]_\mu$ is therefore parametrized by such a μ. Correlating syntactic substitution $\mathtt{M}\{X \mapsto \mathtt{A}\}$ and semantic modification of the machine environment $\mu(X \mapsto [\![\mathtt{A}]\!]_\mu)$ gives then rise to substitution theorems on the level of machines and on the level of programs that closely resembles the one of predicate logic, i.e.$[\![e]\!](s(x \mapsto a)) = [\![e\{x \mapsto e'\}]\!](s)$ for $a = [\![e']\!](s)$.

While this view does not seem lead to a shortcut of some kind for the proofs of compositionality, the approach renders the results in a potentially more familiar way. With a semantics that depends on machine environments, substitution can characterized by

$$\begin{aligned} I(\underline{x} \mapsto I(\underline{ax})) \in runs^{[\![\mathtt{M}]\!]_{\mu(X \mapsto [\![\mathtt{A}]\!]_\mu)}} \quad &\Longleftrightarrow \quad I \in runs^{[\![\mathtt{M}\{X \mapsto \mathtt{A}\}]\!]_\mu} \\ I(\underline{x} \mapsto I(\underline{ax})), \mu(X \mapsto [\![\mathtt{A}]\!]_\mu) \models p \quad &\Longleftrightarrow \quad I, \mu \models p\{X \mapsto \mathtt{A}\}, \end{aligned}$$

where it assumed that the variables $\underline{x}$ capturing the state of X are known. The modified interval $I(\underline{x} \mapsto I(\underline{ax}))$ swaps in the whole trace $I(\underline{ax})$ of `A` as an assignment to $\underline{x}$. In this setting, a submachine call $\mathtt{Op}^X(\underline{e}; \underline{z})$ occurring in p refers to the atomic semantics $[\![\mathtt{Op}^{\mu(X)}]\!]$ of the operation of the machine bound to X.

Remark. The concept of variables ranging over machines is not as uncommon as it may sound, see for example their use in ambient ASMs [30]. Of course, the fundamentals of object-oriented languages are closely related, although in this thesis, the system's structure

is not dynamic, whereas typically calculi for object-oriented programs rely on simulation-like proof obligations (e.g., Hatcliff et al. [75], Liskov and Wing [101]) instead of inclusion of runs.

Chapter 6

Models in Flashix

3. Das logische Bild der Tatsachen ist der Gedanke.
— Ludwig Wittgenstein

Summary. This chapter gives a brief technical overview over the data structures and models in the refinement tower of Flashix, starting with the top-level POSIX specification and going down to the interface of the persistence layer. Along the way, it is shown how the concepts outlined in the introduction of this thesis are realized and how they can be modeled formally and abstractly.

Publications: This chapter is based on [144].

The next chapters bring together the conceptual description of Chapter 2 and the theoretical foundations of Chapter 4 and 5 by detailing the formal models and technical decomposition of the Flashix system. The way how crash-safety is realized is postponed to Chapter 12. The full details can be found in the web-presentation [49]. The models document not only the the effort in general to develop a verified file system, but more importantly, how concepts are captured abstractly and decomposed incrementally towards the implementation.

The formal models are described systematically by giving their state space, the operations, and their invariants as we have seen already in the Examples 4.12 and 5.5. The following concrete notation and conventions are used. Program variables and logical variables are written in *italics*, algebraic constants, functions, and predicates are written in `typewriter` font, type names are capitalized and *`slanted`*, and keywords are written in **`boldface`**.

The state space of a machine `M` is by listing the state variables and their types as

`state vars` (`M`) $x_1 : St_1, \cdots, x_n : St_n$.

We generally use the keyword **`state vars`** for implementation level machines (i.e., the ones from which the running code is derived), contrasted by the keyword **`spec vars`**, which highlights that the machine corresponds to an abstract specification model that is not compiled into the code. The distinction does not indicate a technical difference, though. Initial states and invariants of a machine `M` are specified syntactically by

`initial state` (`M`) $\varphi(\underline{x})$
`invariant` (`M`) $\psi(\underline{x})$

meaning that $Init^{\mathtt{M}}(\underline{x}) \coloneqq \varphi(\underline{x})$, where φ and ψ are a formula over the state variables $\underline{x} = x_1, \cdots, x_n$ of M. A named operation is prefixed with the respective machine and is specified by

M_op($\underline{in}$; $\underline{out}$)
  precondition $\varphi(\underline{in}, \underline{x})$
  body

with formal input parameters $\underline{in}$ and output parameters $\underline{out}$ separated by a semicolon ";", a precondition φ, and an ASM program as body. For simplicity, the state of the respective model is omitted in the parameter lists. In the concrete KIV models, however, this state is made explicit since KIV does not admit global variables, and consequently the state variables x_i accessed by a particular operation are part of the formal reference parameters (resp. as part of the input parameters when the state is only read but not modified).

This chapter and the next one progress by outlining a series of refinement steps $\mathtt{A} \sqsubseteq \mathtt{C}$ from a specification A towards a composite implementation $\mathtt{C} \coloneqq \mathtt{M}(\mathtt{X})$ that refers to a submachine X. For the next refinement step X recursivelly takes the role of the specification of the subpart of the respective development.

Typically, each implementation level M is introduced alongside the abstract specifications of its subcomponents X: To understand the implementation of M operations and their integration with X it is necessary to know how X works internally. The coupling between the state spaces in a refinement step (relation R in Section 5.2) is consequently specified as

coupling $(\mathtt{A} \sqsubseteq \mathtt{M}(\mathtt{X}))$ $\varphi(\underline{ax}, \underline{mx}, \underline{xx})$,

where formula φ may refer to the state variables of all three machines.

When defining the state space of any abstract (sub-)machine A two orthogonal goals are followed. On one hand, it should represent the problem domain subject to this machine as abstractly as possible. This facilitates reasoning about its invariants and likewise about the behavior of its context M. On the other hand the state A may be enriched with auxiliary state that is not strictly necessary to specify functionality but to shift much of the proof burden for invariants and the refinement coupling towards the abstract layer.

This section complements Section 2.2 from a technical side. The general pattern is to gradually switch from a nested, functional encodings towards flat, imperative concepts such as records, pointers, and arrays. The monolithic view of the state in terms of tree-like algebraic data types is thereby broken up in favor of smaller quantities that can be modified individually and are linked together by indirections. Examples are a graph-based view at the level of VFS in contrast to the POSIX model which is a proper tree and the index in the file system's core that references on-flash data.

An excerpt of the model stack from Figure 2.2 in the introduction is reconsidered as shown in Figure 6.1. We to briefly summarize the different concepts and where they are addressed.

The abstract POSIX model detailed in Chapter 7 at the top of the hierarchy captures the requirements of the file system.

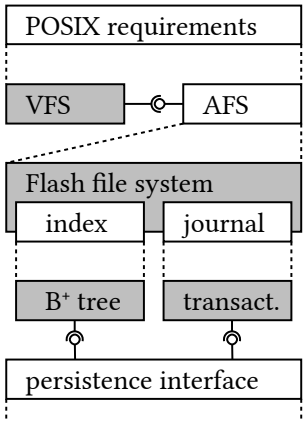


Figure 6.1: Upper Layers of Flashix.

It deals with the concepts described in the textual POSIX specification [3] resp. the manual pages available on UNIX systems: the directory tree, files, paths, access rights, and file handles.

This specification is realized modularly by the combination of the Virtual File System (VFS) and the Abstract File System (AFS) specification in Chapter 8. Generic aspects shared by all concrete file system implementations are handled in VFS, without relying on a specific representation on disk. The two are integrated by an interface and a common data model.

As a consequence of this separation of concerns in Chapter 8, the flash specific part of the project can focus on a subset of the challenges. The flash file system as described in Chapter 9 realizes strategties to efficiently deal with the specific characteristics of the flash hardware. It uses two subcomponents: The journal mediates access to the underlying storage and provides atomicity in the presence of power cuts. The index tracks the location of current versions of data on flash, which necessarily moves around due to the no-overwrite restriction of the storage technology.

Both subcomponents of the file system core access the flash memory through the persistence layer, which determines the device layout and offers byte-based access to various regions storing different kinds of data structures. The remaining components are not subject to this thesis and will not be described.

Example 6.1 (Directory Tree)**.** On a Unix system, for example, the file system hierarchy can be explored by the `tree` shell command:

```
> tree /home
home
├── ernst
│   └── notes.txt
│   └── references.bib
...
```

The command follows the prompt `>`, the output appears in the subsequent lines. In the example, there is the user's home directory with some notes and a bibliography file.

Paths are formed by concatenating directory and file names with a separating character "`/`" following the UNIX convention.[1] Paths can be either absolute or relative to a "current directory", although this work deals with absolute paths only. An example absolute path is `/home/ernst`.

Example 6.2 (Creating a File)**.** The running example will be the creation of a new file named `thesis.tex` in the user's home directory. The following C code calling the system operation `creat(e)`[2] accomplishes this task in a POSIX compatible environment:

```
int err = creat("/home/ernst/thesis.tex", 0644);
```

The first parameter denotes the full (absolute) path to the new file and the second parameter denotes an access mode (the octal value `0644` stand for user-writable, world-readable).

To satisfy the request to create a file, the file system looks up the path `/home/ernst` to the parent directory. A new empty file is allocated and an entry pointing to it is recorded in the parent directory.

[1] On Windows, the separator is a backwards slash "\".
[2] The trailing "e" is not part of the name in the actual C POSIX interface.

POSIX (Chapter 7). The representation of the tree in the formal POSIX model is a recursive, functional data structure *tree*: `Tree` with directories (containing subtrees) and files (containing a link to their content). The actual content of files is stored separately in a finite file store *fs*. All modifications to this tree are expressed as non-destructive operators, for instance, lookup of a path p is written $tree[p]$ and replacing the subtree at p with a different one $tree'$ is written $tree[p \mapsto tree']$. The tree thus resembles a mapping from paths to some kind of data, but with a specific internal structure by construction. For the example above the modified state is determined by

$$tree[\texttt{"/home/ernst/thesis.tex"} \mapsto \texttt{fnode}(fid)] \quad \text{and} \quad fs[fid \mapsto \texttt{file}(\texttt{0644}, [\,])]$$

for a new file identifier *fid*, where the file gets the specified mode `0644` and an empty list of bytes $[\,]$ as its content.

Virtual File System (Chapter 8). Within the AFS that stores the state of the next layer of the hierarchy, this monolithic tree is broken up: files and directories are identified by unique "inode" numbers *ino*: `Ino`, following the UNIX naming tradition. Files and directories are kept in two disjoint maps *dirs*: `Ino` $\rightharpoonup$ `Dir` and *files*: `Ino` $\rightharpoonup$ `File` that correspond to the tree *tree* and the file store *fs* of the POSIX model. The tree structure enforced by the *tree* data structure at the POSIX level must now be ensured through explicit invariants as the data model is a pointer-based graph. In the AFS model, objects of `Dir` contain a store `entries`: `String` $\rightharpoonup$ `Ino` that maps names of children to their respective inode numbers. The content of files in is contained in the `File` objects. To continue the example, local modifications after resolving the path are expressed by assignments

$$dirs[pino].\texttt{entries}[\texttt{"thesis.tex"}] := ino \quad \text{and} \quad files[ino] := \texttt{file}(\texttt{0644}, \cdots)$$

where *pino* identifies the parent directory and *ino* the new file.

The connection to the POSIX model is given for a file $ino \in \texttt{dom}(files)$ by a corresponding $fid \in \texttt{dom}(fs)$. For a directory $ino \in \texttt{dom}(dirs)$, the relation is more complex: there is a path $p \in tree$ such that any path p/s extended by a name s induces $s \in \texttt{dom}(\texttt{entries})$ of that directory, such that the correspondence holds recursively for $\texttt{entries}[s]$.

Flash File System Core (Chapter 9). The flash file system core flattens the VFS/AFS representation in favor of three kinds of objects that are already present in the data model of the VFS interface: *inodes* representing (the metadata of) files and directories, *directory entries* encoding the edges of the file system graph, and *pages* storing the content of files. These objects are uniformly encoded into so called "nodes", which are identified uniquely by keys. Yet another indirection is introduced by linking keys to the addresses of the corresponding node on flash. The main data structures that model this view abstractly are the RAM index *ri*: `Key` $\rightharpoonup$ `Address` and the flash store *fs*: `Address` $\rightharpoonup$ `Node` so that lookup of a node corresponding to a key is a two-stage process $adr := ri[key]; nd := fs[adr]$ (the credit for this model goes to Schierl et al. [147]). Note that this encoding of the state does not resemble much the intuition of a hierarchical file system any more. The keys and corresponding nodes that are relevant for creating a file are

$$\begin{aligned} \texttt{inodekey}(pino) &\rightsquigarrow \texttt{inodenode}(\cdots, size + 1, \cdots) \\ \texttt{dentrykey}(pino, \texttt{"thesis.tex"}) &\rightsquigarrow \texttt{dentrynode}(\cdots, ino) \\ \texttt{inodekey}(ino) &\rightsquigarrow \texttt{inodenode}(\cdots, \texttt{0644}, \cdots), \end{aligned} \tag{6.1}$$

where the squiggly arrow $\rightsquigarrow$ informally stands for a mapping via some address. Some bookkeeping information in the parent is updated to account for the new child, too.

The state of the core is related to AFS so that whenever $\texttt{inodekey}(ino) \in ri$ either $ino \in \texttt{dom}(dirs)$ or $ino \in \texttt{dom}(files)$ and the metadata of $fs[ri[\texttt{inodekey}(ino)]]$ matches. For the other types of keys/nodes, the correspondence looks into `Dir` resp. `File` for the respective inode number. Note that such an abstraction is robust against changing the addresses, i.e., the composition $fs \circ ri$ of the flash store and the RAM index already determines the observable behavior of the file system core. This is a desired feature: Moving data around during garbage collection of on-flash memory doesn't leak to the upper layers.

Transactional Journal. Each top-level file system operation such as creating a file will affect several nodes simultaneously. These updates come in groups that must be made power cut safe: only when all of the `Node`s of such a group reach the flash store the operation is considered to have taken effect. Ensuring this is the task of the implementation of the transactional *journal*, which maps the unordered flash store *fs* to the block structure of the flash device. However, blocks are still kept abstract at this level of detail as mathematical sequences of individual `Node` objects. The state—that is part of the specification of the persistence layer—can be roughly characterized by a map $gblocks\colon \texttt{Nat} \rightarrow\!\!\!\!\!\!+\ \ \texttt{List}\langle \texttt{Header} \times \texttt{Node}\rangle$ (group blocks) where the domain contains logical block numbers and each node is accompanied by a header that carries the grouping information.

Appending the three nodes (nd_i below) of (6.1) for the example to $gblock = gblocks[lnum]$ with logical number *lnum* gives an updated state $gblocks[lnum] := gblock'$ for

$$gblock' = gblock \mathbin{+\!\!+} [(\text{start}, nd_1), (\text{mid}, nd_2), (\text{end}, nd_3)],$$

where the "start" and "end" markers signify the group boundaries, and $gblock \mathbin{+\!\!+} [\cdots]$ denotes concatenation of the node group at the end of the previous content of the block.

The connection to the unordered store *fs* of the journal layer is made by making the type $\texttt{Address} = \texttt{LNum} \times \texttt{Nat}$ more concrete. Addresses $adr = (lnum, offset)$ now store a logical block number and a byte-based offset of the group node in that block, which is then mapped to the respective position in the list.

Index Implementation. The implementation of the index is a B^+ tree [18], which is a data structure that can represent large sets or mappings efficiently. There are several aspects that make the implementation complicated: In the context of the Flashix file system, the index implementation has a few further complications in comparison to a standard B^+ tree.

The RAM index is loaded partially only. Its nodes are allocated on demand and populated from the flash copy. Conceptually, this leads to a structure that interleaves parts of the two versions of the index as shown in Figure 6.2. To that purpose, each downwards link does not only point to the node in memory but also stores the address of the corresponding node on flash.

As the RAM index is continuously modified, the mismatch to the outdated version on flash accumulates. This difference is called the *dirty* part of the RAM index. Dirty nodes may not be removed from the cache (in contrast to non-dirty ones). These nodes are marked with a small black box ◥ at the top-right. The dotted lines denote that the corresponding flash node has not been loaded into main memory yet.

Since updating the on-flash copy of the index with each operation is very costly, these are delayed for some time and then written out in batch transactions, so that the index

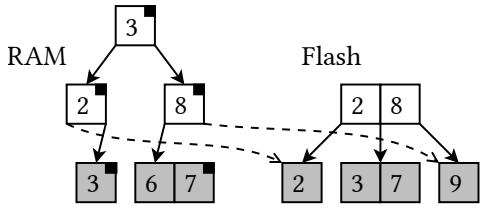


Figure 6.2: RAM & flash B^+ tree.

"wanders" around on the flash device [88]. The parts of the on-flash index that haven't been changed can be reused, though.

Like the nodes encoding file system objects, the nodes making up the structure of the B^+ tree are laid out in the logical flash blocks sequentially in a map *iblocks* ("index blocks") that mirrors *gblocks*.

Uniform Modeling of Hardware Errors A general concern that is present in almost all of the abstract models in the following is to adequately express what errors can occur. The mathematical data types on which the state space is based on are reliable: one can assume infinite, perfect storage so that e.g. reading, writing, and allocation always succeeds. However, transient and persistent failures are quite common with real flash hardware; allocation may fail due to insufficient memory, and the flash device can be full. Since the specification-level models are way too abstract to capture when exactly such errors arise, most operations are permitted to fail nondeterministically. This is achieved by a uniform pattern for operations:

$$\begin{aligned} &\texttt{A\_op}(\cdots; \mathit{err}) \\ &\quad \{\ \mathit{body},\ \ \mathit{err} := \texttt{ESUCCESS}\ \} \\ &\ \ \textbf{or}\ \{\ \textbf{choose}\ \mathit{err} \in \{\texttt{EIO}\ \cdots\}\ \} \end{aligned} \tag{6.2}$$

Either the operation succeeds with the regular effect *body*. In this case, the returned error code *err* indicates success. Otherwise, *err* is selected from a set of low-level errors (including a generic input/output error `EIO`), without modifying the state in any way. The returned error codes are then checked by the caller, and either masked by some alternative code path or propagated further up the model stack potentially to the POSIX application level. Whenever an operation does *not* admit such errors it is explicitly mentioned in the description of the models and furthermore indicated by omitting the output parameter *err*.

Chapter 7

POSIX Model

Specification of Correctness Requirements

Summary. This section presents a concise and intuitive model of the POSIX file system interface that follows the textual standard [3] closely. It captures various corner cases and subtleties on which real programs rely on. The model authoritatively explains how Flashix works from the client's perspective.

Publications: This chapter is based on [50, 51].

Contents

7.1 State

The file system state consists of a directory tree *tree*, a file store *fs*, and a registry of open file handles *oh*. Files are referenced by file identifiers of the abstract sort *Fid*. Open files are referenced by natural numbers ("file descriptors" in Unix).

spec vars (POSIX) $tree\colon Tree$
$fs\colon Fid \pfun FileData$
$oh\colon Nat \pfun Handle$

The directory tree is specified as an algebraic data type *Tree* with two constructors: File nodes (`fnode`) form the leaves and store the identifier of the corresponding file. Directory nodes (`dnode`) make up the internal nodes and store the directory entries as a mapping from names to the respective subtrees.

data $Tree = \texttt{fnode}(\texttt{fid}\colon Fid)$
$\mid \texttt{dnode}(\texttt{meta}\colon MetaData, \texttt{entries}\colon String \pfun Tree)$

The test $t.\texttt{dir}?$ yields whether tree t is a `dnode`. The abstract sort *MetaData* is a placeholder for any further associated information. We postulate some selectors for $md\colon MetaData$, to retrieve for example read, write and execute permissions $\texttt{pr}(user, md)$, $\texttt{pw}(user, md)$,

$\texttt{px}(user, md)$ for some unspecified user $user\colon \mathit{User}$. This formalization of permissions has been taken from Hesselink and Lali [81].

Files are given by the data type $\mathit{FileData}$ that stores the content as a list of bytes, and—analogously to directories—some associated metadata.

$$\textbf{data}\ \mathit{FileData} \;=\; \texttt{fdata}(\texttt{meta}\colon \mathit{MetaData}, \texttt{content}\colon \mathit{List}\langle \mathit{Byte}\rangle)$$

File handles store a file identifier `fid`, and keep track of the current read/write offset `pos` in bytes, and a mode, which can be read-only, write-only or read-write.

$$\begin{aligned}\textbf{data}\ \mathit{Handle} \;&=\; \texttt{fhandle}(\texttt{fid}\colon \mathit{Fid}, \texttt{pos}\colon \mathit{Nat}, \texttt{mode}\colon \mathit{Mode})\\ \mathit{Mode} \;&=\; \texttt{r} \mid \texttt{w} \mid \texttt{rw}\end{aligned} \tag{7.1}$$

The initial state is given by an empty root directory and no files:

$$\textbf{initial state}\ (\text{POSIX}) \quad tree = \texttt{dnode}(md, \varnothing) \wedge fs = \varnothing \wedge oh = \varnothing \tag{7.2}$$

7.2 Path Lookup and Tree Modifications

A directory tree $t\colon \mathit{Tree}$ is indexed by paths $p\colon \mathit{List}\langle \mathit{String}\rangle$, which are lists of names (strings). The symbol ϵ denotes the empty path and $/$ denotes concatenation of paths p, p' resp. paths and individual segments $s\colon \mathit{String}$, written s/p, p/s, and p/p' respectively.

The directory tree can be regarded as a partial function from paths to subtrees enriched with some internal structure. Consequently, the notation to index a tree t is borrowed from the one for partial functions: A path p is valid in a directory tree t, written $p \in t$, if starting from the root t the path can be followed recursively such that each path segment is mapped by the respective subdirectory. Similarly, lookup of a valid path $p \in t$ is written $t[p]$. The expression $t[p \mapsto t']$ denotes the tree t where the subtree t' has been stored at path p, possibly replacing an existing subtree; modification is only defined if the parent of p is a directory. Conversely, $t - p$ denotes the tree t without the whole subtree at path p; it is only defined if $p \in t$. Finally, we write $p \sqsubseteq p'$ when the path p is a (non-strict) prefix of p'.

It should be emphasized that the update operations $t[p \mapsto t']$ and $t - p$ are functional: a new tree is constructed as a result instead of destructively modifying the old one. This means that one does not have to worry about aliasing, sharing, or cycles.

The four operators validity, lookup, update, and deletion are the foundations on which the ASM specification of the POSIX interface is based on (Figure 7.2).

Definition 7.1 (Path Operators)**.** Validity and lookup of paths is defined by the following recursive equations. Lookup at a file node is meaningless and therefore left un(der)specified.

$$\begin{aligned}\epsilon \in t &\leftrightarrow \texttt{true}\\ s/p \in \texttt{fnode}(fid) &\leftrightarrow \texttt{false}\\ s/p \in \texttt{dnode}(md, st) &\leftrightarrow s \in st \wedge p \in st[s]\end{aligned}$$

$$\begin{aligned}t[\epsilon] &= t\\ \texttt{dnode}(md, st)[s/p] &= st[s][p] \quad \text{if } s \in st\end{aligned}$$

The first line states that the empty path ϵ is valid in every tree. A compound path s/p with leading name s is valid in directories only, if s denotes a valid subtree in st containing the remainder p recursively.

Addition and deletion of paths can be done only at directories, the cases discern whether the path consists of one component s only, or whether one has to recurse for a remainder p, in which case $s \in st \wedge p \neq \epsilon$ is presumed (second/fourth line).

$$\begin{aligned}
&\texttt{dnode}(md, st)[s \mapsto t'] &&= \texttt{dnode}(md, st[s \mapsto t'])\\
&\texttt{dnode}(md, st)[s/p \mapsto t'] &&= \texttt{dnode}(md, st[s \mapsto st[s][p \mapsto t']])\\
&\texttt{dnode}(md, st) - s &&= \texttt{dnode}(md, st - s)\\
&\texttt{dnode}(md, st) - s/p &&= \texttt{dnode}(md, st[s \mapsto (st[s] - p)])
\end{aligned}$$

Directory trees $t : \mathit{Tree}$ have several nice structural properties related to paths. Validity of paths is prefix-closed, i.e., if $p/p' \in t$ then $p \in t$, furthermore $t[p]$ is a directory node if $p' \neq \epsilon$. Validity, lookup, insertion and deletion of paths compose with path concatenation $\_\,/\,\_$, for example:

$$\begin{aligned}
&p/p' \in t \leftrightarrow (t \in p \wedge p' \in t[p])\\
&t[p/p'] = t[p][p'] \quad \text{if } p \in t
\end{aligned} \tag{7.3}$$

These properties can be proved easily by induction over the length of the paths.

7.3 Operations

Operations realize the POSIX specification by using the algebraic functions on trees. The POSIX model is presented in Figure 7.2—omitting some precondition checks and generic error handling code at the beginning of each operation, which is explained in Section 7.4. For the moment, assume that the input satisfies the constraints outlined in the `man`-pages, e.g., that a path passed to `create` does not exist yet.

Figure 7.1 shows an abstract graphical representation of a POSIX file system. The directory part forming a proper tree is visualized as in triangular form at the top. Each individual directory corresponds to a subtree, i.e., the structure is inherently recursive, where each directory is the root of a subpart of the file system hierarchy. The grey part of Figure 7.1 represents the hierarchy under `/home/ernst`.

To continue Example 6.2, the operation `create` receives the path *path* to the new file and its mode as part of the abstract metadata *md*. An identifier *fid* for the file is chosen that is not already in use. Subsequently, both the tree is modified to contain an `fnode` at the given path using the algebraic operations defined in the previous section, where the assignment $t[p] := t'$ abbreviates $t := t[p \mapsto t']$ analogously to function update. The new file is stored in *fs* with the given metadata and an empty sequence $[\,]$ of bytes as its content. The effect of the operation is visualized in Figure 7.1. The grey subtree corresponds to the parent directory $t[\texttt{parent}(p)]$; the newly created file node and associated data are denoted by the dashed triangle and box respectively.

In Figure 7.1 the effect of the create operation is depicted by a dotted contour. The grey triangle denotes the subtree with the parent directory as its root, and the small white triangle denotes the new entry named `thesis.tex`. The entry for the new file in the tree points to the content of the new file at the bottom. This extra indirection supports hard-links, i.e., several references to the same file under different names.

The operations in Figure 7.2 are divided into structural ones shown in the left column and access to file content shown in the right column.

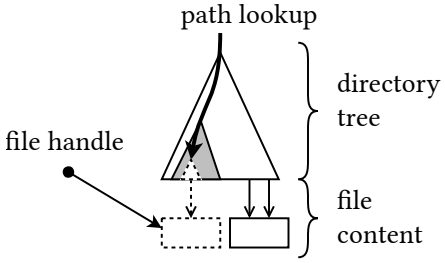


Figure 7.1: Directory Tree.

```
posix_create(path, md; err)
  choose fid with fid ∉ fs
  in   tree[path] := fnode(fid)
       fs[fid]    := fdata(md, [])

posix_link(from, to; err)
  let fid = tree[from].fid
  in   tree[to] := fnode(fid)

posix_unlink(path; err)
  let fid = tree[path].fid
  in   tree := tree − path
       if fid ∉ fids(tree, oh)
       then fs := fs − fid

posix_rename(from, to; err)
  let t1 = tree[from],   t2 = tree[to]
       exists = (to ∈ tree)
  in   tree := tree − to
       tree[to] := t1
       if     exists ∧ ¬ t2.dir?
          ∧  t2.fid ∉ fids(tree, oh)
       then fs := fs − t2.fid

posix_mkdir(path, md; err)
  tree[path] := dnode(md, ∅)

posix_rmdir(path; err)
  tree := tree − path

posix_readmeta(path; md, err)
  if tree[path].dir?
  then md := tree[path].meta
  else md := fs[tree[path].fid].meta

posix_writemeta(path, md; err)
  if tree[path].dir?
  then tree[path].meta := md
  else fs[tree[path].fid].meta := md

posix_readdir(path; names, err)
  names := dom(tree[path].entries)

posix_open(path, mode; fd, err)
  let fid = tree[path].fid
  in   choose n with n ∉ oh
  in   fd  := n
       oh[fd] := fhandle(fid, 0, mode)

posix_close(fd; err)
  let fid = oh[fd].fid
  in   oh := oh − fd
       if fid ∉ fids(tree, oh)
       then fs := fs − fid

posix_read(fd; buf, len, err)
  let fid = oh[fd].fid
       pos = oh[fd].pos
  in   choose n with n ≤ len
       in len := n
       copy(fs[fid].content, pos, 0, len; buf)
       oh[fd].pos := pos + len

posix_write(fd, buf; len, err)
  let fid = oh[fd].fid
       pos = oh[fd].pos
  in   choose n with n ≤ len
       in len := n
       splice(buf, 0, pos, len; fs[fid].content)
       oh[fd].pos := pos + len

posix_truncate(path, len; err)
  let fid = tree[path].fid
  in   resize(len; fs[fid.content])
```

Figure 7.2: Operations in the POSIX model (omitting error handling).

The operation `link` and `unlink` create resp. remove hard links on existing files. Linking a file under a different path simply amounts to adding a corresponding leaf to the tree that carries the (unique) identifier of the target. The converse operation must additionally check if the last reference to the file has just been deleted, in which case the content must be freed as well (deletion from *fs*)

Directories are created by `mkdir`, which stores a new node with an empty set of subtrees at the given path under the tree. Since directories cannot be hard-linked, the removal case is easier: `rmdir` simply removes the node, although POSIX requires such directories to be empty.

The most complex structural operation is `rename`, which allows to change the name and optionally the parent directory of a file or directory, potentially overwriting the existing file/directory at the target path (under some restrictions). It accesses and modifies the tree at two different paths and has several cases that combine the effect of the other structural modifications. The reason why `rename` is a separate operation, though, is that the whole modification of the file system's state can be implemented *atomically*. This feature of POSIX is relied on by applications for data consistency, since it can be used to overwrite a file virtually atomically by first preparing the new version and then replacing the old version with `rename`.

Access to the metadata of files and directories is provided by `readmeta` and `writemeta`, which subsume `stat`, `chmod`, `chown`, and so on. The operation `readdir` lists the contents of a given directory by the names of the direct children (from the domain of the subtree mapping).

File content is accessed through file handles that are obtained for a specific mode (read, write, or both) by `open` and released by `close`. Opening a file chooses an unallocated descriptor *fd* that is initialized to a handle at the beginning of the file (position 0). Closing a handle must check whether the file is an had been unlinked from the tree before and if so deallocate the file's content when no other file handles remain.

Reading and writing through a file handle transfers a slice of bytes between the input resp. output buffer *buf* where *len* denotes the desired length of the slice. As the operations `read` and `write` can transfer large amounts of data in one call, it is neither practical nor required by the textual standard to provide atomicity of these operations (even disregarding power cuts). Instead, the operations may process less than *len* bytes and still *succeed*, either because the concrete implementation runs out of disk-space during the write, or due to an intermediate low-level error. This is modeled by nondeterministically setting *len* to a smaller number ("short read and write").

When the reason for a short read or write persists then the next attempt is expected to fail (in the case of a full disk-space). Other reasons that could be recovered later on include temporary storage pressure that is in the process of being resolved (e.g. by background garbage collection), in which case an implementation may indicate the special error code `EAGAIN`.

Lastly, the operation `truncate` permits one to change the size of an existing file, padding the content with zero-bytes at the end when the file grows.

The ASM code relies on several helpers that operate on lists of bytes: `resize`(*len*; *l*) adjusts the size of list *l* to *len*, possibly padding *l* with zeroes at the end; `copy` and `splice` copy *len* elements of the source list *src* starting from offset *src-pos* into list *dst* at offset *dst-pos*, preserving the elements outside of the destination range. While `copy` keeps the size of *dst*, `splice` may increase the size as necessary. The latter operation corresponds

```
posix_create(path, md; err)
  choose err with pre-create(path, md, tree, fs, err)
  if err = ESUCCESS
  then choose fid with fid ∉ fs
       in tree[path] := fnode(fid)
          fs[fid]    := fdata(md, [])
```

Figure 7.3: The create operation of the POSIX model without simplifications.

exactly to the semantics of the POSIX write operation, i.e., it may extend *dst* at the end as shown below, where $\#l$ denotes the length of l.

```
splice(src, src-pos, dst-pos, len; dst)
  if len ≠ 0 then
     if dst-pos + len < #dst
     then resize(dst-pos + len; dst)
     else copy(src, src-pos, dst-pos, len; dst)
```

7.4 Preconditions and Error Handling

All operations perform extensive *error checks* to guard the file system against unintended or malicious calls to operations. Specifically, all operations are total (defined for all possible values of input parameters). However, there are requirements that must be fulfilled in order for an operation to succeed. For example, to create a new file, the given path must not refer to an existing one already. The term "prerequisite" will be used in the following to characterize valid inputs to contrast such conditions from proper preconditions, where a violation leads to an unspecified effect. However, violation of prerequisites here must lead to an error *without* observably modifying the state. This behavior is required by POSIX and it has been acknowledged by some formal approaches also [81], whereas others that model errors leave the outcome open [57]. The nondeterministic approach presented below was introduced by Ernst et al. [51], though.

Insufficient handling of the returned error codes of C system calls is a well-known source of problems that possibly have security implications, since the program(mer) relies on assumptions that can be invalidated. An attacker may deliberately provoke error codes, for example by exhausting the resources of the machine. Similarly, correct error handling inside the Flashix file system causes a lot of additional effort. Internal error conditions are either mitigated locally by the implementation or propagated to the POSIX surface

Error handling is nondeterministic. It is possible that two errors conditions hold simultaneously, e.g., the whole path does not exist, or permissions to traverse an existing prefix are insufficient. The POSIX specification does not restrict the order in which different conjuncts of prerequisites are checked. Prerequisites are defined as predicates pre-*op*$(in, \cdots, err)$ that specify possible error codes *err* for an input *in* given to the operation *op*. An implementation just has to satisfy the constraints imposed by these predicates.

Figure 7.3 shows the full ASM code of the `create` operation in the model. Initially, the error code *err* is chosen so that complies with the prerequisite predicate. The effect is only executed if the operation is bound to succeed (implying a valid input).

Prerequisite-predicates contribute a significant part of the specification, just like documentation of the many corner cases takes up most of the standard. The predicates are

defined by case distinction on possible error codes, as exemplified below for the condition `pre-create`. Success is only admitted (case $err = \texttt{ESUCCESS}$) when the path to the new file does not exist yet, the parent is fact a directory, and when the user has sufficient permissions to access and write to the directory at $\texttt{parent}(path)$ (omitted). The remaining cases take apart these conjuncts and attach potential error codes to them. Certain errors, such as hardware failure or memory allocation (denoted by $\texttt{EIO}, \cdots$) are not restricted, i.e. they may occur anytime. All other errors cannot occur (last case).

$$\texttt{pre-create}(path, md, tree, fs, err)$$

$$\leftrightarrow \begin{cases} path \notin tree \wedge \texttt{parent}(path) \in tree \wedge tree[\texttt{parent}(path)].\texttt{dir?} \\ \wedge \text{ permissions hold} & \text{if } err = \texttt{ESUCCESS} \\ path = \epsilon, & \text{if } err = \texttt{EACCESS} \\ \text{(insufficient permissions)} & \text{if } err = \texttt{EACCESS} \\ path \in tree, & \text{if } err = \texttt{EEXIST} \\ \texttt{parent}(path) \notin tree, & \text{if } err = \texttt{ENOENT} \\ \neg\, tree[\texttt{parent}(path)].\texttt{dir?}, & \text{if } err = \texttt{ENOTDIR} \\ \texttt{true}, & \text{if } err \in \{\texttt{EIO}, \cdots\} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

The precise mechanism how the permission checks are specified is described in detail by Hesselink and Lali [81] and won't be repeated here.

Recovering from low-level errors $\texttt{EIO}, \cdots$ in the implementation without an observable state change is by far trickier than detecting whether the prerequisites are satisfied. In fact, in Chapter 8 it will be shown how almost all of the high-level errors that can arise by violation of prerequisites can be dealt with uniformly and generically as part of the Virtual File System (VFS), without considering any details specific to a concrete implementation or flash memory.

7.5 Invariants

The POSIX model maintains two explicit invariants over the state with tree *tree*, file store *fs* and open file handles *oh*. The easy one is simply that the root must be a directory The second invariant states that the set of file identifiers referenced by *tree* or *oh* is equal to $\texttt{dom}(fs)$. It guarantees that for any *fid* in use, the associated file data in *fs* is available (no dangling hard-links—well-definedness of lookups $fs[tree[path].\texttt{fid}]$ in Figure 7.2), and that *fs* contains no garbage (all obsolete files are in fact cleaned up):

`invariants` (POSIX)

$$tree.\texttt{dir?} \quad \wedge \quad \texttt{dom}(fs) = \texttt{fids}(tree) \cup \texttt{fids}(oh), \tag{7.4}$$

given two overloaded functions to determine file identifiers in the tree resp. in the store of open handles. The first one is defined recursively over the structure of the tree:

$$\texttt{fids}\colon \mathit{Tree} \to \mathit{Multiset}\langle \mathit{Fid}\rangle$$
$$\texttt{fids}(\texttt{fnode}(\mathit{fid})) = \{\!|\mathit{fid}|\!\}$$
$$\texttt{fids}(\texttt{dnode}(\mathit{md}, \mathit{st})) = \biguplus_{s \in st} \texttt{fids}(\mathit{st}[s]).$$

File nodes produce a singleton multiset with that one identifier. For directory nodes the subtrees are accumulated ($\uplus$ is multiset sum). The second one is just the identifiers in the range of *oh*:

$$\texttt{fids}\colon (\mathit{Nat} \rightharpoonup \mathit{Handle}) \to \mathit{Set}\langle \mathit{Fid}\rangle$$
$$\texttt{fids}(\mathit{oh}) = \{\mathit{oh}[n].\texttt{fid} \mid n \in \mathit{oh}\}.$$

Multisets are preferred over ordinary sets for the file identifiers in the tree for two reasons. On one hand, the number of occurrences of *fid* in the set $\texttt{fids}(t)$ correlates with the number of hard links to a file. On the other hand, the effect of insertion or removal of a subtree on `fids` directly maps to multiset sum $\uplus$ and difference $\setminus$, respectively. Given an existing path to a parent directory node $p \in t$, equations (7.5) and (7.6) can be proven by structural induction on p.

$$\texttt{fids}(t[p/s \mapsto t']) = \begin{cases} \texttt{fids}(t) \uplus \texttt{fids}(t') & \text{if } p/s \notin t \\ \texttt{fids}(t) \uplus \texttt{fids}(t') \setminus \texttt{fids}(t[p]) & \text{otherwise} \end{cases} \tag{7.5}$$

$$\texttt{fids}(t - p) = \texttt{fids}(t) \setminus \texttt{fids}(t[p]) \tag{7.6}$$

With these lemmas, the proofs for invariant (7.4) are straight forward when taking the prerequisites of the operations into account. The critical cases are in `unlink`, `rename`, and `close`, where the check occurs whether the last link or file handle to an orphan has just been removed.

7.6 Orphans and Power Cuts

Files that are referenced only as open files but not from the directory tree are called *orphans*. These arise when a file that is currently open loses its last link due to an `unlink` call.

Orphan files are useful during package upgrades, for example, when the binary file of a running application is overwritten. The running process still keeps a pointer to the old version of the file and prevents the file system from deleting its contents on disk. Orphans are also used as hidden lock files in the Apache web server and to store temporary data outside of the visible name space by the MySQL database. To achieve this the application deletes a file immediately after opening it. POSIX requires that the content of an orphaned file can be read and written normally until the file is closed.

The file system tracks these references precisely to know when a file's content can actually be released, either upon deletion of the last hard-link (operation `unlink`) or upon closing the last file handle (operation `close`). Formally, the set of file identifiers of orphaned files can be captured by

$$\texttt{orphans}(t, \mathit{fs}) = \texttt{dom}(\mathit{fs}) \setminus \texttt{fids}(t).$$

Another characterization is based on the function

$$\texttt{links}(\mathit{fid}, t) = \{\ p \in t \mid t[p] = \texttt{fnode}(\mathit{fid})\ \}$$

that tells, *which* links to a file exist as a set of absolute paths in the directory tree. From invariant (7.4) it is easy to see that $\texttt{links}(\mathit{fid}, t) = \varnothing$ exactly when $\mathit{fid} \in \texttt{orphans}(t, \mathit{fs})$.

Orphans demonstrate already in this abstract setting that an understanding of crash tolerance as observable atomicity of operations is insufficient: when the system crashes, data stored in main memory is erased, in particular, all processes running at that point in time cease to exist and the opportunity to delete orphans during a `close` operation will be missed, yet there are no remaining references. Therefore during recovery the file system must be burdened with deleting all existing orphans.

Technically, the effect of a crash is characterized by a binary relation over pairs of states (from unprimed to primed). For the POSIX model, it is assumed that both the directory tree and the file content are stored persistently, so that they remain unchanged:

$$\textbf{crash}\ (\text{POSIX}) \quad \mathit{tree}' = \mathit{tree} \wedge \mathit{fs}' = \mathit{fs} \tag{7.7}$$

The open file handles in contrast are stored in main memory and the value oh' after a crash is thereby completely unconstrained. The recovery operation, which has to reconstruct a valid state after a crash, therefore reinitializes oh and deletes the orphaned files:

$$\begin{aligned}&\texttt{posix\_recover}(\,)\\ &\quad \mathit{oh} := \varnothing\\ &\quad \mathit{fs} := \mathit{fs} \setminus \texttt{orphans}(\mathit{tree}, \mathit{fs})\end{aligned}$$

Note that after recovery, invariant (7.4) from Section 7.5 is reestablished.

Observe that this abstract specification does not give so much indication of how to *implement* such a requirement. Specifically, not losing any data as stated by (7.7) is a hard task. It is also up to the implementation to determine the set $\texttt{orphans}(t, \mathit{fs})$ in terms of its internal data structures. A direct computation from the complete state (e.g. by scanning the entire device) would be infeasibly slow, so in practice the set is maintained explicitly.

The theory that integrates crashes and recovery into the refinement method of this thesis will be explained in Chapter 11. How the requirements outlined in this section are implemented and verified in the flash file system is postponed to Chapter 12.

7.7 Related Work

In the literature there are many efforts in the area to file system modeling and verification with different scope and data structures that can be related do different levels of abstraction and models in the refinement chain described in this chapter. This section focuses mostly on the *modeling aspects* of normal behavior, in contrast to treatment of power cuts and the verification. Comparison to related work regarding these two aspects is done in detail later on.

In the context of NASA's challenge [91], Freitas et al. [60] list a number of issues to be addressed: adequacy of the specification of the POSIX interface and the subset modeled, consistency of data structures after unexpected power-cuts and dealing with the unreliability of the flash memory hardware itself (i.e., fault tolerance in general), allocation, even use, and reclamation of storage space. The paper aims to integrate the challenge with a

proposal by Intel [1], which defines APIs for flash access and file systems at many different levels of abstraction. In this thesis, the approach is not followed, as the document [1] is fairly large and seems to impose some unjustified overhead. Nevertheless, the conceptual difficulties are summarized well in [60].

The existing modeling and proof efforts related to file systems can roughly be classified into two categories:

The first category encompasses high-level models that capture isolated aspects at a high-degree of abstraction only, where it is sometimes unclear or even doubtful that the presented formalizations can serve as top-level specification for a *realistic* and running implementation. Much of the work in this category can be linked to NASA's challenge [91] and related activity in the Verified Software Repository [158]. However, none of the efforts has come near an implementation that actually provides a working file system. Much of this work is a continuation from an early pen-and-paper model of the POSIX file system interface by Morgan and Sufrin [108], written in the specification language Z [159]. Nevertheless, the models in this category give a lot of details about different approaches and thereby have built a solid foundation for successive work. A survey of this work has been published by Lali [98].

The second category encompasses full-blown developments that typically have the goal to produce running code and take additional challenges into account. For example, Chen et al. [34] have developed a simple but working file system for conventional magnetic storage and proved it correct and power-cut safe. Such recent results clearly demonstrate the continued interest in systems software verification with a focus on realistic problem sizes that are by now in the reach of automated theorem provers.

In the literature there are several approaches to modeling the directory tree of a POSIX file store.

The approach to formalize a POSIX file system with an algebraic tree as has been used previously only by Heisel [78] to evaluate specification languages and specification reuse, which is an orthogonal interest as it doesn't measure the ease of proofs.

The work of [108] is based on a flat mapping from paths to files and directories. This approach has been picked up e.g. by the mechanizations of Ferreira et al. [57], Hesselink and Lali [81]. It comes at the cost of an extra invariant that path validity is prefix-closed ("legitimate stores") that is verified in Hesselink and Lali [81]. The invariant holds by construction in this work:

Proposition 7.2 (Legitimate stores.)**.** *Directory trees* t: `Tree` *model legitimate stores as a direct consequence of* (7.3)*:*

$$\epsilon \in t \quad \wedge \quad \forall\, p, q.\ p \sqsubseteq q \wedge q \in t \rightarrow p \in t$$

A lot more concepts are described in Morgan and Sufrin [108], such as hard-links, orphaned files, and reading and writing through file handles. While it covers a wide range of POSIX related topics it is not mechanized and consequently does not have a strong focus on proofs (of invariants for instance). The work has partly been mechanized by Freitas et al. [61] in the Z-Eves theorem prover (although it does not consider paths at all).

Except for [132], there seems to be no formalization of hard links that is mechanized. In Hesselink and Lali [81], equivalence classes of paths are suggested as an alternative solution. We are not aware of an attempt to realize this idea, though it would be interesting.

Damchoom et al. [40], in contrast, formalize the hierarchical structure by parent pointers with an acyclicity invariant. Hard links are inherently not supported by this design. We

think that this approach is too different from the intuitive understanding of a file system to serve as top-level specification.

Prerequisites that characterize valid inputs to POSIX operations are treated similarly to Hesselink and Lali [81], i.e., operations must not modify the state on errors. Ferreira et al. [57] also have a comprehensive error specification in their POSIX-style specification, however, they fix the order of checks and allow arbitrary behavior on errors in their refinement proof obligations (although they haven't actually developed refined models). To our knowledge, underspecified hardware failures are admitted elsewhere only in [132].

Morgan and Sufrin's work [108] describes read and write operations on sequences of bytes as the model presented in this chapter does. It contains a minor error (that has been taken over by [61, 92]): they do not specify an equivalent of the test $len \neq 0$ in `splice` (Section 7.3), which may result in overly large files. The corresponding requirement in the POSIX standard [3] states that "[...] if nbytes [=len] is zero [...] the `write()` function shall return zero and have no other results." Instead, the `write` operation is specified as follows

$$content \coloneqq \texttt{zero}(pos) \oplus content \oplus (buf \ \texttt{shift} \ pos)$$

where $\texttt{zero}(pos)$ is an file of length pos, which introduces the padding if $pos \geq \#content$. The operator $\oplus$ denotes relational override in Z, and `shift` adjust the input buffer buf from 0-based to pos-based indices. As a consequence, a write of length 0 beyond the current size of the file will always increase it needlessly. The models in [65, 132] are similar in this regard, as the descriptions of the POSIX read and write operations are expressed at the level of sequences of bytes (although the error above is not present in these works).

Curiously this error had also been present in the early stages of the formal POSIX model in Flashix. It was discovered during failed refinement proofs when the implementation did in fact handle this situation correctly. A corresponding test case has been integrated into SibylFS [132] by the suggestion of the author of this thesis.

Chapter 8

Virtual File System

Separating Generic File System Concepts

Summary. This section shows, how the composed system of VFS —©—AFS breaks down the top-level POSIX concepts down to local operations and a representation of the file system as a graph structure. The interplay between VFS and AFS that has been outlined in Section 2.2.2 is made precise. It is shown how the formal models separate different concerns.

Publications: This chapter is based on [50, 51].

Contents

The combination VFS —©—AFS realizes the POSIX interface in a first formal refinement step. Conceptually, the VFS component breaks the global, connected view of a file system in terms of *tree* and *fs* in favor of a local understanding of directories and files as individual entities and nodes of a graph structure. The edges of the graph corresponding to the entries in directories are made more explicit as well. The graph based view in Figure 8.1 contrasts to the "monolithic" directory tree object of the POSIX model shown in Figure 7.1. On a technical level, this switch amounts to the introduction of an indirection, i.e., the underlying data model becomes a pointer structure. The nodes—called *index nodes* (inodes) for historic reasons—are identified by unique *inode numbers*, and the edges—called *directory entries* (dentries)—are identified uniquely by the inode number of the parent and a name. File content, which is attached to the leaves in the tree, is partitioned into uniformly sized segments—called VFS *pages*—that match the size of the virtual memory pages of the CPU architecture (typically 4 KB) to support efficient caching coordinated by the VFS (which is already prepared here and currently being realized outside the scope of the thesis).

The graph structure permits one to map the algorithmic steps needed to perform a system-level POSIX operation to multiple AFS operations. Figure 8.2 shows the interaction between the two layers in terms of the call sequence for Example 6.2 that deals with creating the file `/home/ernst/thesis.tex`. Each segment of the path to the parent direc-

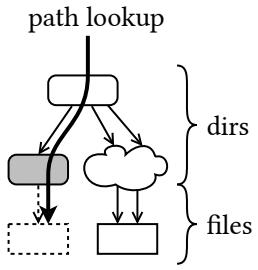


Figure 8.1: Directory Tree as a pointer structure.

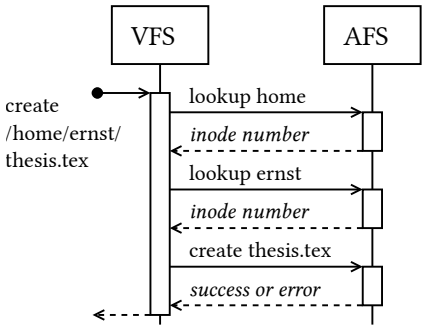


Figure 8.2: Interplay between VFS and AFS.

tory `/home/ernst` is looked up individually yielding the respective inode number. Along the way, VFS checks whether the current user has the access rights to traverse the given path and whether he may in fact create the file (not shown). Finally a request to create the new file is issued that returns an error or indicates success. In Figure 8.1 the new file and the edge representing the corresponding directory entry are depicted by a dotted contour. The parent directory is marked in grey.

8.1 Data Model and Abstract File System Interface

The focus of this section is the interface between VFS and AFS, more specifically, the view that the VFS has in terms of its *data model*. The high-level integration is outlined, while more specific aspects are addressed in subsequent sections. Since the VFS layer is an implementation level component that will end up as executable code, the data structures passed over this interface are concrete as well, moreover, they should be independent of (but close to) the actual representation of data on flash resp. on disk. The file system's state is a graph encoded by three types of objects, namely index nodes (inodes) representing files and directories, directory entries dentries (dentries) with names attached to them, and pages containing uniformly segments of file data.

Inodes correspond to the nodes of the graph, i.e., the files and directories. Inodes are uniquely identified by an inode number $ino\colon Ino \simeq Nat$ and store some associated information. The sort *Inode* is formally defined as an algebraic data type:

$$\begin{aligned}\textbf{data}\ Inode\ = \texttt{inode}(&\texttt{ino}\colon Ino,\ \texttt{meta}\colon MetaData,\ \texttt{dir?}\colon Bool,\\ &\texttt{nlink}\colon Nat,\ \texttt{size}\colon Nat,\ \texttt{nsubdirs}\colon Nat)\end{aligned}$$

It has one constructor `inode` that records the inode number (`ino`), some metadata (`meta`) such as permissions and timestamps, whether it corresponds to a file or a directory (`dir?`), the number of hard-links (inbound edges, `nlink`), the file size resp. the number of directory entries in case of a directory inode (`size`). In the latter case, it is tracked how many of these are again directories (`nsubdirs`).

Dentries correspond to the edges of the graph. They relate one of the directories to its children (represented by their inode numbers) and are labelled with the respective *file names*. Dentries store a `name` and come in two flavors: Normal dentries point to an existing file identified by the selector `target`. Negative dentries indicate that a file name is *not* contained within a directory—they are used for example as return value of the lookup

operation.

data *Dentry* = dentry(name: *String*, target: *Ino*)
| negdentry(name: *String*)

The content of files is partitioned into uniformly sized *pages*. This has several advantages: The size of pages typically corresponds to the size of a virtual memory page, enabling caching and memory-mapped input/output. Furthermore, sparse files, i.e. files with large empty parts, can be represented efficiently by the convention that non-present pages contain zeros only. Pages are modeled as arrays of bytes of a fixed length, specified by the constant `VFS_PAGE_SIZE`:

type *Page* = $Array_{\texttt{VFS\_PAGE\_SIZE}}\langle Byte \rangle$,

where the empty page containing all zero bytes is denoted by $[0, \cdots, 0]$.

Since these data structures are not (necessarily) stored, they serve primarily for *communication* purposes. Either the file system implementation or the VFS creates a number of these on-demand in order pass information back and forth over the interface —◎—between the two layers.

The signature of the AFS operations is modeled by its counterpart in the Linux kernel, just like the data structures are. For example, the two operations `lookup` and `create` involved in creating a file are implemented by file systems in the Linux kernel by functions of the following types:

```
struct dentry *lookup(struct inode *dir, struct dentry *dent, ...);
int create(struct inode *dir, struct dentry *dent, int mode, ...);
```

Both operations receive the inode `dir` to the parent directory and a negative dentry with the name of the link, which is then populated with the target of the lookup resp. the new file by the concrete FS implementation.

In the formal model, these are mapped to ASM operations with similar signatures:

afs_lookup(*inode*: *Inode*; *dent*: *Dentry*, *err*: *Error*)

afs_create(*md*: *MetaData*; *inode*: *Inode*, *dent*: *Dentry*, *err*: *Error*)

Note that *dent* is a reference parameter in both operations (coming after the semicolon), mirroring the fact that the directory entry passed in is being modified to contain the target's inode number as the result of the operation. The `create` operation additionally modifies the parent *inode* by increasing its size by one to accommodate the new entry in that directory. The abstract metadata *md* captures the `mode` of the Linux C interface. Both operations also return an error code *err* indicating whether the operation had been successful.

The operations of the AFS have true preconditions in the sense that it is assumed that the VFS already validates the user-supplied input and thereby establishes the prerequisites outlined for the top-level POSIX operations in Section 7.4.

Remark. Although many AFS operations take full-blown objects of type *Inode*, sometimes it is convenient or even necessary to indicate just the inode number (e.g. when the VFS obtains an *Inode* in the first place). At the time of writing, the internal interface between the two layers is being refactored to simplify the introduction of uniform caching at the level of the three communication data structures and consequently favors passing these. The presentation in the publications [50, 51], in contrast, is typically just based on the inode numbers for simplicity.

8.2 State

Having defined the communication data structures so far, this section presents, how these are maintained internally by the AFS as part of its state. Recall that while this state is hidden behind an interface during runtime, it is necessary for the understanding of the overall behavior of the composed system VFS ⊸◎⊸ AFS and to relate it to the counterpart of the POSIX model of Chapter 7.

AFS maintains as its internal state two separate stores for files and directories mapping inode numbers to the respective objects, encoding the directory tree *tree* and the file store *fs* from the POSIX level as an equivalent pointer structure.

spec vars (AFS) $\quad dirs\colon Ino \rightharpoonup Dir, \quad files\colon Ino \rightharpoonup File.$

Note that there is no preimposed requirement to keep these in two separate stores. The separation is motivated by the distinction into structural and content modifications: the former will affect mainly *dirs* while the latter will affect only *files*. This decision simplifies the refinement proofs between the POSIX layer and VFS (and makes a sum-type $Dir + File$ unnecessary). However, it comes at the cost of an extra simple disjointness invariant (see Section 8.7). Inode numbers $ino \in (\mathtt{dom}(dirs) \cup \mathtt{dom}(files))$ are called *allocated*, they refer to valid directories resp. files.

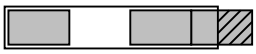
The data types for files and directories are as defined follows:

$$\begin{aligned}&\textbf{data}\ Dir \;=\; \mathtt{dir}(\mathtt{meta}\colon MetaData,\ \mathtt{size}\colon Nat,\ \mathtt{nlink}\colon Nat,\\ &\qquad\qquad \mathtt{nsubdirs}\colon Nat,\ \mathtt{entries}\colon String \rightharpoonup Ino)\\ &\textbf{data}\ File = \mathtt{file}(\mathtt{meta}\colon MetaData,\ \mathtt{size}\colon Nat,\ \mathtt{nlink}\colon Nat,\\ &\qquad\qquad \mathtt{content}\colon Nat \rightharpoonup Page)\end{aligned}$$

The respective first part of the constructor arguments reflects the information stored in the *Inode*s of the VFS data model. Although this information is partially redundant, it turned out to be a good idea to introduce the redundancy early on in the refinement chain, where it is much easier to prove the associated consistency invariants.

At the heart of the definitions are the maps `entries` resp. `content`, where the directory entries and pages of the data model are represented. These are contained in the enclosing object they belong to, i.e., the entire information about one file or directory is kept as one object (cf. the introduction to this chapter). Files explicitly track their sizes, because that cannot be recovered from the allocated pages alone (which are aligned to `VFS_PAGE_SIZE`).

The store `content` of pages can be *sparse* by simply omitting any page that contains just zeroes. This feature has the potential of saving a lot of disk space for applications (e.g. exploited by backup tools, databases, and for disk images by hardware emulators such as QEMU and VMWare). In the following, a file's content is visualized in terms of the pages surrounded by a box denoting the relevant part in the range $0 \cdots \mathtt{size}$:


AFS pages

The grey boxes represent allocated pages. The empty space in the middle represents an absent page implicitly containing zeroes. The hatched part at the end denotes the part of the last page that is cut off by the size of the file (surrounding box). This part does not contain any relevant bytes, however, as pages are uniformly sized, it is still there. The

```
vfs_create(path, md; err)
  if   path = ϵ then
       err := EACCESS
  else let
       ino  = ROOT_INO
       dent = negdentry(path.last)
       path = path.parent
  in   vfs_walk(path; ino, err)
       if err = ESUCCESS then
          vfs_may_create(ino; inode, dent, err)
       if err = ESUCCESS then
          afs_create(md; inode, dent, err)
```

```
vfs_walk(path; ino, err)
  err := ESUCCESS
  while path ≠ ϵ ∧ err = ESUCCESS do
    vfs_may_lookup(ino; err)
    if err = ESUCCESS then
       let dent = negdentry(path.head)
       in  afs_lookup(ino; dent, err)
           if err = ESUCCESS then
              ino  := dent.target
              path := path.tail
```

Figure 8.3: Operations of the VFS/AFS model realizing file creation, prefixed by the respective layer they belong to. Permission checks are done by the `*_may_*` routines. Lookup of a single path segment is delegated to `afs_lookup` filling in the target in the reference parameter *dent*.

abstract view as a linear sequence of bytes can be recovered by concatenating the pages and zeroes for the empty spaces up to the file's size (see Section 8.8).

The combination of *dirs* and *files* represents the tree t and the file store *fs* of the POSIX model of Chapter 7. The remaining constituent of the state, namely the store of open file handles, is managed by the VFS analogously to POSIX, except that it is now understood as part of the implementation.

state vars (VFS) $oh : \mathit{Nat} \rightarrow\!\!\!\!\!\!+\ \mathit{Handle}$ where

data $\mathit{Handle} = \mathtt{fhandle}(\mathtt{ino}: \mathit{Ino}, \mathtt{pos}: \mathit{Nat}, \mathtt{mode}: \mathit{Mode})$

where file handles are equivalent to (7.1) except for the fact that they refer to inode numbers *Ino* instead of file identifiers *Fid*.[1]

The initial state is given by an empty root directory with a fixed inode number `ROOT_INO` and no files:

initial state (VFS/AFS)

$$\mathit{dirs} = [\mathtt{ROOT\_INO} \mapsto \mathtt{dir}(\mathit{md}, \varnothing)] \wedge \mathit{files} = \varnothing \wedge \mathit{oh} = \varnothing \tag{8.1}$$

8.3 Structural Operations

The structural VFS operations (cf. Section 7.3) all follow the same pattern: a path walk by `vfs_walk` followed by the corresponding local AFS operation.

To continue the Example 6.2, the VFS code to create a file is shown in Figure 8.3 alongside the operation `vfs_walk` for path traversal. The entry point is `vfs_create`, receiving the path *path* to the new file and its meta data *md*. The first line already picks up the extensive error handling seen in Section 7.3: an empty path ϵ leads to an error `EACCESS`. Starting at the root directory, referred to by `ROOT_INO`, lookup of the path excluding its last segment is initiated by a call to `vfs_walk`. For each step the permission are checked first and a (negative) dentry with the name of the current segment is prepared and the result of the lookup

[1] The refinement relation shown in Section 8.8 identifies these two types $\mathit{Ino} \simeq \mathit{Fid}$.

```
afs_lookup(pino; dent, err)
  precondition pino ∈ dirs
  if dent.name ∈ dirs[pino].entries
  then let ino = dirs[pino].entries[dent.name]
       in  dent := dentry(dent.name, ino)
           err  := ESUCCESS
  else dent := negdentry(dent.name)
       err  := ENOENT
```

```
afs_create(md; inode, dent, err)
  precondition
       inode.ino ∈ dirs
     ∧ dent.negdentry?
     ∧ dent.name ∉ dirs[pino].entries
  let pino = inode.ino in
  choose ino ≠ 0 with ino ∉ dirs ∧ ino ∉ files
  in dirs[pino].entries[dent.name] := ino
     dirs[pino].size += 1
     files[ino] = file(md, 0, 1, ∅)
     dent := dentry(dent.name, ino)
     err := ESUCCESS
  or { choose err ∈ {EIO, ···} }
```

Figure 8.4: AFS code for lookup and create. Nondeterministic errors are only sketched for `create`. In the full model these are integrated as shown in (6.2) into both operations, as both do access the potentially faulty flash storage.

is found in *dent*.`target` after the call to `afs_lookup`. At that time, the prerequisite of the operation has been established and thus the precondition of `afs_create` is satisfied. Back in `vfs_create` the file is created using the directory entry prepared with the actual name *path*.`last` of the file. Note that the *inode* of the parent is returned by the `vfs_may_create` helper as an optimization (which needs it anyway to check the permissions) so that it can be passed directly to `afs_create`.

The AFS counterpart to Figure 8.3 is shown in Figure 8.4. The `lookup` operation checks whether the indicated file name is stored in the parent directory specified by *pino*. If so, the target inode number *ino* is returned as part of an updated positive directory entry and success is indicated. Otherwise, a negative directory entry is returned alongside the error code `ENOENT`. The create operation needs to select a fresh identifier *ino* different from 0 (zero inode numbers are not valid) that is not in use for an existing directory or file. The entries of the parent directory *pino* are updated with the given additional mapping and the content of the file is created as an object with the provided metadata *md*, a size of 0, a link count of 1, and an empty store of pages. The returned directory entry is populated with the new inode number.

The other structural operations `link`/`unlink`, `mkdir`/`rmdir`, and `rename` are likewise realized by one or two traversals of the directory tree and a subsequent local AFS operation.

Note that the `create` operation shown in Figure 8.4 admits potential errors of the hardware in the last line, following the pattern (6.2) for abstract specifications of this aspect.

8.4 Deletion

As described in Section 7.3, files become obsolete on two occasions, namely in `unlink`, `rename`, and `close` when the last hard-link resp. file handle is dropped. While in the POSIX model this was easy to accomplish because all the information was readily available from the state, the knowledge about file handles and hard-links is now separated: the VFS knows the former (from *oh*) whereas AFS knows the latter (from *dirs*).

The two layers synchronize by a protocol where VFS signals an explicit call to the operation `afs_evict` whenever the last handle is closed by `vfs_close`, implemented by

```
vfs_close(fd; err)
  if ¬ fd ∈ dom(oh)
  then err := EBADF
  else oh := oh − fd
       vfs_putinode(ino)

vfs_putinode(ino)
  if ¬ is-open(ino, oh)
  then afs_evict(ino)

afs_evict(ino)
  if links(ino, dirs) = ∅ then
     if ino ∈ dirs then dirs := dirs − ino
     if ino ∈ files then files := files − ino
```

Figure 8.5: Two-step protocol for eviction.

the helper procedure `vfs_putinode`. The three operations are shown in Figure 8.5, where the predicate

$$\texttt{is-open}(ino, oh) \leftrightarrow \exists\, fd \in oh.\ oh[fd].\texttt{ino} = ino$$

determines whether there is still some file handle *fd* for the given *ino*. Note that directories are deleted through `afs_evict` as well for uniformity, which is called unconditionally from `vfs_rmdir` right after the corresponding local `afs_rmdir`. The set of links

$$\texttt{links}(ino, dirs) = \{(pino, name) \mid dirs[pino].\texttt{entries}[name] = ino\}$$

to an inode with number *ino* is computed from the directory store as pairs $(pino, name)$ such that *pino* has a directory entry *name* pointing to the file/directory *ino*.

One particular aspect with eviction is that it may never fail as indicated by the absence of an error output. The reason is that the state has already been modified at that point in time by the previous flash operation (`afs_unlink`, `afs_rmdir`, or `afs_rename`) and thus the corresponding VFS call may not fail as well to satisfy the requirement that only successful operations can have observable effects as specified by the formal POSIX model in Section 7.4. This requirement is reasonable, because `afs_evict` will be implemented by an in-memory operation. In Linux, this restriction is imposed also (the `void` return type of `evict` prevents signalling an error).

8.5 File Truncation

The operation `truncate` changes the size of a file to a specified value, potentially extending it with zeroes at the end. In the page-based view of AFS this means that one has to pay attention to the last page in the array (cf. Section 8.2) that contains additional bytes: whenever the range extends to include a fraction of this part, the last page must be rewritten. Alternatively, one might attempt to state the invariant that these bytes are already cleared, however, as discussed later on, in the presence of hardware errors (and later on power-cuts), this invariant becomes impossible to maintain. It is easier to fix up the situation during truncate (UBIFS does the same).

The code of the AFS model is shown in Figure 8.6. First, the number and the offset of the potentially affected page are computed. In the critical case, the page is patched by copying `VFS_PAGE_SIZE` − *offset* bytes from the empty page.

```
afs_truncate(inode, len; err)
  precondition inode.ino ∈ files ∧   // the fields of inode match the data stored at files[ino]
  let content = files[ino].content
      pageno  = inode.size / VFS_PAGE_SIZE
      offset  = inode.size % VFS_PAGE_SIZE
  in  if inode.size ≤ len ∧ offset ≠ 0 ∧ pageno ∈ content
      then copy([0, ···, 0], offset, offset, VFS_PAGE_SIZE − offset; content[pageno])
      files[ino].content := content upto len
      files[ino].size := len
```

Figure 8.6: File truncation in AFS.

```
afs_readpage(ino, pageno; page, err)
  precondition ino ∈ files
  let content = files[ino].content
  in  if   pageno ∈ content
      then page := content[pageno]
      else page := [0, ···, 0]
```

```
afs_writepage(ino, pageno, page; err)
  precondition ino ∈ files
  files[ino].content[pageno] := page
```

Figure 8.7: Access of single pages.

The algebraic operation *content* `upto` *len* deletes mappings $n \mapsto page$ from *content* that fall outside the new file size, i.e., where the first byte in the page n is at least *len*.

$$content \texttt{ upto } len = content \setminus [(n \mapsto page) \mid n \cdot \texttt{VFS\_PAGE\_SIZE} \geq len]. \tag{8.2}$$

8.6 Reading and Writing

The basic building blocks for the operations `vfs_read` and `vfs_write` are two AFS operations that access a single page in the store *content*: `Nat` ⇸ `Page` of a given file. These are shown in Figure 8.7.

Reading and writing in VFS maps a linear buffer onto the file's array of pages. There is a number of corner conditions and special cases that must be considered, rendering both the implementation inside VFS as well as the refinement proofs quite complicated:

- Files may contain holes (unallocated pages in the `content` of a `file` object) that implicitly represent a part of the file filled with zeroes. These empty pages are reconstructed on the fly by the AFS internally and lead to some case distinctions (cf. `afs_readpage` in Figure 8.7)
- Reading and writing transfers ranges of bytes that are not necessarily aligned at page boundaries. The input/output buffer passed to the `read` resp. `write` operation is therefore filled incrementally with chunks that come from subranges of pages, where the first and last page accessed must be treated specially. The arithmetic involves division and modulo and is therefore nonlinear which is undecidable in general.
- A write operation may implicitly increase the size of a file when the transferred range reaches over the end of the file. This means that a) the part of the last page that contains some bytes beyond the original file size suddenly become relevant and it matters at which point in time these are zeroed out and b) together with an

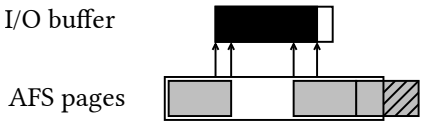


Figure 8.8: Reading unallocated pages.

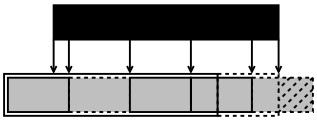


Figure 8.9: Extending the size.

extending write the new size of the file must be written to flash storage, which may fail due to an error or an interruption by a power cut and it is not immediately clear how that can be done in accordance with the specification given in Figure 7.2.

- As an outlook on the verification: The different cases encountered in the loop body of the algorithm shown below tend to accumulate quadratically in the invariants of the proofs (see Section 8.8).

These aspects and the way they are addressed will be explained in more detail below by graphical illustrations and the concrete ASM code.

An example read operation is visualized in Figure 8.8. At the bottom, the file's pages are denoted by gray boxes with an outer frame indicating the file size. The hatched part of the last page contains arbitrary data and does not contribute to the file's content. The white space among the pages denotes an unallocated page, which implicitly represents a range of all zeros in the file. The rectangle at the top denotes the destination buffer. The black part corresponds to the range that should be read. The buffer may be larger than this range (white part). The read operation loads the affected pages sequentially and copies the required parts (graphically delimited by arrows) into the buffer.

Listing 8.10 shows the core helper procedure that is called in a loop until done or an error occurs. Parameters *start* and *end* define the range to read, as absolute positions into the file in bytes. Of this range, *total* bytes have been processed so far, the operation therefore considers a subrange starting at position $pos = start + total$ of length $n \leq$ `VFS_PAGE_SIZE` bytes. Note that *done*, *buf* and *total* are passed by reference. The procedure computes the current page and offset into that page and considers three upper bounds for the length n of the range to copy in this iteration: the maximum number of bytes to transfer (*len*), the end of the current page, and the end of the file. Nonexistent pages are handled by `afs_readpage` already as shown in Figure 8.7.

Writing is done similarly, an example is shown in Figure 8.9, assuming the operation is executed with the same file as in Figure 8.8. A write operation may extend the file at the end. In this case, the dotted lines indicate the newly allocated parts of the file: the missing second page is written, as well as an additional page at the end. The fourth page, which contained the irrelevant hatched part, is overwritten and becomes part of the file entirely. The write operation relies on a helper routine similar to `vfs_read_block` (not shown). The changes for `vfs_write_block` amount to leaving out the inode size boundary in the computation of the number of transferred bytes n, also dropping the precondition $start + total \leq inode.\texttt{size}$ for being able to extend the file size. Naturally, the direction of the `copy` call is reversed, followed by a `afs_writepage` with the updated contents of the page.

The VFS code of the operation `vfs_read` calling the one-step procedure `vfs_read_block` is shown in Figure 8.11. After a few error checks to ensure that the file descriptor *fd* is set up for reading, the inode object of the file is loaded (`afs_iget`) and the range *start* to *end* is initialized, starting with a total number of 0 bytes processed so far. Everything else is

```
vfs_read_block(start, end, inode, buf, dirs; total, done, err)
  precondition   inode.ino ∈ files ∧ ¬ done ∧ err = ESUCCESS
                 ∧ start + total ≤ inode.size ∧ total ≤ #buf
  let pos    = start + total
      pageno = pos / VFS_PAGE_SIZE
      offset = pos % VFS_PAGE_SIZE
  in  let  // bytes to read in this iteration
          n = min(end − pos,                  // write size boundary
                  VFS_PAGE_SIZE − offset,     // current page boundary
                  inode.size − pos)           // inode size boundary
  in  if n > 0 then
          afs_readpage(inode.ino, pageno; page, err)
          if err = ESUCCESS then
              copy(page, offset, total, n; buf)
              total := total + n
      else done := true
```

Figure 8.10: Reading a subrange of a partial page.

```
vfs_read(fd; buf, len, err)
  if fd ∉ oh then err := EBADF
  ...  // check read mode of fd
  afs_iget(oh[fd].ino; ino, err)    // load file information
  if   err = ESUCCESS then
  let  start = oh[fd].pos,   end = start + len
       total = 0,   done = false in
  if   start ≤ inode.size then
       while ¬ done ∧ err = ESUCCESS do
             vfs_read_block(start, end, inode, buf, dirs; total, done, err)
       if total ≠ 0 then err := ESUCCESS
       oh[fd].pos := oh[fd].pos + total
       len := total
  else len := 0
```

Figure 8.11: Implementation of the `read` operation in VFS.

handled by the subroutine, including advancing the counter *total* and setting the *done* flag when the transfer is complete.

Upon an error that can arise from a failed page read the transfer is interrupted (loop check for $err = \texttt{ESUCCESS}$), but the error is ignored if any bytes have been processed at all (check for $total \neq 0$): the reason is that the read operation so far was indeed successful and it is adequate return the partial result as discussed in Section 7.4. Ultimately, the current read offset in the file handle is advanced and the number of bytes copied into *buf* is returned in the output parameter *len*.

The `vfs_write` implementation is rather similar to `vfs_read`. After initialization which potentially zeroes out a last page that now becomes part of the file using `afs_truncate`, all the bytes in the range are transferred with the subroutine `vfs_write_block`. The critical question is when to write the new size of the file in case it must be increased so that it is robust against the failure to write *all* of the pages. Writing the new size before even starting out means that the size will grow too much but instead of the bytes proper the tail

of the file contains zeroes or even garbage if some obsolete pages were previously present and the initial truncate fails. This is not admitted by the POSIX specification. Writing the new size in lockstep with the pages is inefficient and just wastes a lot of space for the metadata updates of the file that needs to be reclaimed later on.

Writing the new size at the end when it is known how far the operation has succeeded is therefore the only option, but it may well be possible that this ultimate change to the storage medium fails, too. This strategy in fact complies with the POSIX requirements: regardless whether the old or the new size is stored on flash, the bytes up to this value will be the right ones, and the whole operation can *succeed* by restricting *len* to whatever fits into the file as given.

8.7 Invariants

We have proved the following simple invariants on the VFS and AFS state *oh* resp. *dirs* and *files*. Inode numbers 0 are never used (8.3) as they are used by the FFS core to denote deletion in some cases, conversely, the distinguished constant `ROOT_INO`: *Ino* is a directory (8.4). The domains of *dirs* and *files* are disjoint (8.5). Invariant (8.6) states closure under lookup: following a directory entry leads to an allocated inode number. Directories have at most one parent (8.7). The inode number of each open file handle is valid (8.8).

invariants (AFS)

$$0 \notin \mathtt{dom}(\mathit{dirs}) \wedge 0 \notin \mathtt{dom}(\mathit{files}) \tag{8.3}$$

$$\mathtt{ROOT\_INO} \in \mathtt{dom}(\mathit{dirs}) \tag{8.4}$$

$$\mathtt{dom}(\mathit{dirs}) \cap \mathtt{dom}(\mathit{files}) = \varnothing \tag{8.5}$$

$$\forall\ \mathit{ino}.\ \mathtt{ran}(\mathit{dirs}[\mathit{ino}].\mathtt{entries}) \subseteq \mathtt{dom}(\mathit{dirs}) \cup \mathtt{dom}(\mathit{files}) \tag{8.6}$$

$$\forall\ \mathit{ino} \in \mathtt{dom}(\mathit{dirs}).\ \mathit{dirs}[\mathit{ino}].\mathtt{nlink} \leq 1 \tag{8.7}$$

invariant (VFS)

$$\forall\ \mathit{fd} \in \mathtt{dom}(\mathit{oh}).\ \mathit{oh}[\mathit{fd}].\mathtt{ino} \in \mathit{files} \tag{8.8}$$

The proofs are not difficult, a couple of helper lemmas lead to high automation. The verification of the AFS invariants can be done locally and crucially relies on the AFS preconditions. For example, the `link` operation requires the target to be a file, otherwise invariant (8.7) may be violated. The `rmdir` operation requires that the path is not empty, so that the root directory is never deleted (invariant (8.4)).

In addition, the redundant fields of files resp. directories must store the correct value that is given by the cardinality of a specific set. For a directory $\mathit{ino} \in \mathtt{dom}(\mathit{dirs})$:

invariants (AFS)

$$\mathit{dirs}[\mathit{ino}].\mathtt{nlink} = |\mathtt{links}(\mathit{ino}, \mathit{dirs})|$$

$$\mathit{dirs}[\mathit{ino}].\mathtt{size} = |\mathtt{dom}(\mathit{dirs}[\mathit{ino}].\mathtt{entries})|$$

$$\mathit{dirs}[\mathit{ino}].\mathtt{nsubdirs} = |\mathtt{dom}(\mathit{dirs}[\mathit{ino}].\mathtt{entries}) \cap \mathtt{dom}(\mathit{dirs})|$$

and for a file $\mathit{ino} \in \mathtt{dom}(\mathit{files})$:

$$\mathit{files}[\mathit{ino}].\mathtt{nlink} = |\mathtt{links}(\mathit{ino}, \mathit{dirs})|$$

8.8 Verification

This section sketches the key steps in the verification of the Virtual File System Switch wrt. its specification, the POSIX model from Chapter 7, i.e., we prove POSIX $\sqsubseteq$ VFS(AFS). The presentation encompasses the definition of the abstraction relation. Separation logic (see Section 3.5) is used to map the pointer-based graph structure in VFS to the algebraic directory tree of POSIX. File content is abstracted to infinite streams instead of finite sequences of bytes, which keeps any argument about file sizes out of the main part of the proofs for reading and writing.

8.8.1 Abstraction to POSIX

The coupling relation R is defined as

coupling (POSIX $\sqsubseteq$ VFS(AFS))
$fs = \mathtt{fs}(files) \ \wedge \ \mathtt{tree}(tree, \mathtt{ROOT\_INO})(dirs)$

where $\mathtt{fs} : (\mathit{Ino} \rightharpoonup \mathit{File}) \to (\mathit{Fid} \rightharpoonup \mathit{File})$ specifies the abstract file store fs and $\mathtt{tree} : \mathit{Tree} \times \mathit{Ino} \to ((\mathit{Ino} \rightharpoonup \mathit{Dir}) \to \mathit{Bool})$ abstracts the pointer structure with root ROOT_INO to the directory tree *tree* using separation logic. By defining $\mathit{Fid} := \mathit{Ino}$, open file handles *oh* coincide in both layers. This section formally defines tree and fs.

Directory Tree. The directory tree is mapped to the store of directories *dirs*, instantiating the separation logic theory from Section 3.5 with $\mathit{Loc} := \mathit{Ino}$ and $\mathit{Val} := \mathit{Dir}$. We define the predicate $\mathtt{tree}(tree, ino)$ by structural recursion on the tree. The idea is that whenever $\mathtt{tree}(tree, ino)(dirs)$ holds, *ino* is the number of the root inode of a file system tree in *dirs* that corresponds to *tree*. We'll use the convention to denote stores for directory entries in the algebraic tree by *st* and to denote stores in the inode based AFS model by *si*. Names of entries are abbreviated by $n : \mathit{String}$ to shorten the presentation to shorten the presentation.

$$\mathtt{tree}(\mathtt{fnode}(fid), ino) \leftrightarrow (\mathtt{emp} \wedge ino = fid) \tag{8.9}$$

$$\mathtt{tree}(\mathtt{dnode}(md, st), ino) \leftrightarrow \tag{8.10}$$
$$\exists\, si.\ \mathtt{dom}(si) = \mathtt{dom}(st) \ \wedge \ ino \mapsto \mathtt{dir}(md, si) \ \star \ \circledast_{n \in st} \mathtt{tree}(st[n], si[n])$$

Assertion (8.9) for file nodes requires that the inode number corresponds to the *fid* of the node and that the remaining part of the heap is empty.

Assertion (8.10) for directory nodes requires a corresponding directory in *dirs* that has the same metadata and corresponding directory entries *si*. The iterated separating conjunction $\circledast$ recursively asserts the abstraction relation for all subtrees $st[n]$ to children $si[n]$ in pairwise disjoints parts of *dirs*.

One can show by induction on p that $\mathtt{tree}(t, ino)(dirs)$ implies

$$\mathtt{path}(p, ino, dirs, files) \ \leftrightarrow \ p \in t$$

We furthermore define the assertion $\mathtt{tree}|_p(t, ino_1, ino_2)(dirs)$ that cuts out the subtree with root ino_2 at path p. Equality (8.11) encodes one main reasoning step for the proofs. It allows us to unfold the directory that is modified by an operation at path $p \in t$:

$$\mathtt{tree}(t, ino_1)(dirs) \ \leftrightarrow \ (\mathtt{tree}|_p(t, ino_1, ino_2) \star \mathtt{tree}(t[p], ino_2))(dirs) \tag{8.11}$$

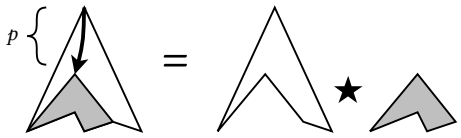


Figure 8.12: Unfolding the tree abstraction at path p.

This lemma is visualized in Figure 8.12.

Another critical lemma discards algebraic tree modifications if p is a (not necessarily strict) prefix of q. It permits to change the grey part in Figure 8.12 without affecting the algebraic part represented in white.

$$q = p/p' \;\rightarrow\; \texttt{tree}|_p(t[q \mapsto t'], ino_1, ino_2) = \texttt{tree}|_p(t, ino_1, ino_2) \tag{8.12}$$

Finally, the abstraction implies the following equivalence, which ensures correct deletion of file content in `close`, `unlink` and `rename`:

$$fid \notin \texttt{fids}(t) \leftrightarrow \texttt{links}(ino, dirs) = \varnothing \quad \text{for } ino = fid$$

File Store. The abstract file store is defined for each $fid \in files, fid = ino$ with $files[ino] = \texttt{file}(md, size, \cdots, content)$ by the extensional equation

$$\texttt{fs}(files)[fid] = \texttt{fdata}(md,\ \texttt{concat}(content)\ \texttt{to}\ size)$$

where $\texttt{concat} : (\texttt{Nat} \nrightarrow \texttt{Page}) \rightarrow \texttt{Stream}\langle\texttt{Byte}\rangle$ assembles a stream of bytes from the pages of a file. The abstract file must store the finite prefix of length *size* of that stream. The abstraction to streams eliminates a lot of reasoning about list bounds and many case distinctions that would otherwise be necessary in definitions and proofs. In particular it simplifies the invariants of the loops in operations `vfs_read` and `vfs_write` in Section 8.8.2.

We define the content of a file as an infinite stream with trailing zeroes beyond the end of the file:

$$\begin{aligned}\texttt{concat}(content) &= \lambda\, n.\, \texttt{getpage}(content, n/\texttt{VFS\_PAGE\_SIZE})[n\%\texttt{VFS\_PAGE\_SIZE}]\\ \texttt{getpage}(content, m) &= \text{if } m \in content \text{ then } content[m] \text{ else } [0, \cdots, 0]\end{aligned}$$

8.8.2 Refinement Proofs

The refinement is proved by forward simulation (Theorem 5.4).

Proof obligation "initialization" is trivial: (8.1) implies (7.2) for the same metadata *md* of the root directory and all invariants hold.

Proof obligation "correctness" is established by symbolic execution of the VFS operation, which yields a state $dirs', files', oh_1', out_1$, followed by symbolic execution of the POSIX operation to construct a matching witness run with a final state t', fs', oh_2', out_2, similar to Example 4.18.

During symbolic execution, whenever the VFS chooses some value by the left rule for $\langle\!|\_|\!\rangle\_$ in (8.13), the POSIX is free to choose the *same* value by the existential quantifier in the right rule for $\langle\_\rangle\_$ in (8.13).

$$\frac{\vdash \forall x.\varphi(x) \rightarrow \langle\!|p|\!\rangle\psi \quad \vdash \exists x.\varphi(x)}{\vdash \langle\!|\texttt{choose}\ x\ \texttt{with}\ \varphi(x)\ \texttt{in}\ p|\!\rangle\psi} \qquad \frac{\vdash \exists x.\varphi(x) \wedge \langle p\rangle\psi}{\vdash \langle\texttt{choose}\ x\ \texttt{with}\ \varphi(x)\ \texttt{in}\ p\rangle\psi} \tag{8.13}$$

The error code err' selected by POSIX is determined this way, as well as e.g. the *fid* in in the operation `create` in Figure 7.3 corresponding to the new inode number *ino* picked in Figure 8.3.

The predicate logic goals resulting from symbolic execution have the form

$$\Gamma \vdash \mathtt{R}(t', fs', oh_1', dirs', files', oh_2') \wedge out_1 = out_2$$

where $\Gamma = \mathtt{R}(t, fs, oh_1, dirs, files, oh_2), \cdots$ contains the initial instance of the simulation relation, as well as preconditions and other information that has been gathered during symbolic execution (e.g., results of the tests in conditionals and subroutine postconditions). The goals reduce to two core proof obligations:

$$\begin{array}{ll} \text{directories:} & \mathtt{tree}(t, \mathtt{ROOT\_INO})(dirs), \Gamma \vdash \mathtt{tree}(t', \mathtt{ROOT\_INO})(dirs') \\ \text{files:} & fs = \mathtt{fs}(files), \Gamma \vdash fs' = \mathtt{fs}(files') \end{array}$$

Proof Strategy for Directories. Two types of modifications to the directory tree occur: insertions $t' = t[p \mapsto \cdots]$ and deletions $t' = t - p$ at a path p. These correspond to a local modification of some directory $dirs' = dirs[ino \mapsto \mathtt{dir}(md', si')]$ (for some new metadata md' and directory entries si') resp. $dirs' = dirs - ino$, where *ino* is found at $\mathtt{parent}(p)$.

The proof strategy is determined by the symbolic execution rules (cf. Section 3.5 for assignment and deallocation. The notation $\psi_h^{h'}$ denotes renaming of the heap h to a fresh variable h' representing the updated heap in the remaining program modality resp. post-condition ψ.

$$\frac{(l \mapsto v \star \varphi)(h') \vdash \psi_h^{h'}}{(l \mapsto \_ \star \varphi)(h) \vdash \langle h[l] := v \rangle \psi} \text{ assign-h} \qquad \frac{(\varphi)(h') \vdash \psi_h^{h'}}{(l \mapsto \_ \star \varphi)(h) \vdash \langle h := h - l \rangle \psi} \text{ dealloc}$$

The first step is to unfold the tree by (8.11) and (8.10) so that the maplet for *ino* is explicit and the assignment can be applied, propagating the assertion to the new directory store $dirs'$. The `dnode` predicate for *ino* is restored wrt. the new subdirectories si', e.g., by introducing an additional `fnode` assertion in the proof for `create`. The context $\mathtt{tree}|_p$ is rewritten to t' as well by (8.12) (applied from right to left), so that the whole abstraction can be folded by reverse-applying (8.11). Most of these steps are automated by rewrite rules.

Proof Strategy for Files. For $ino = fid$ and given $\sigma = \mathtt{concat}(content)$ the following two top-level equalities promote the concrete modification through the abstraction `fs`:

$$\begin{aligned} \mathtt{fs}(files[ino \mapsto \mathtt{file}(size, md, content)]) &= \mathtt{fs}(files)[fid \mapsto \mathtt{fdata}(md, \sigma \ \mathtt{to} \ size)] \\ \mathtt{fs}(files - ino) &= \mathtt{fs}(files) - fid \end{aligned}$$

It remains to establish that σ `to` *size* matches the abstract operation, which is trivial for `create` ($\sigma \ \mathtt{to} \ 0 = \langle\,\rangle$) and difficult for `write` because of the loop in VFS, see Figure 8.9.

The loop invariant for writing states that the file content can be decomposed into parts of the initial file $\mathtt{concat}(content_0)$ at the beginning and at the end, with data from the buffer in between:

$$\begin{array}{lll} \text{write inv:} & \mathtt{concat}(content) = & \mathtt{concat}(content_0) \ \mathtt{to} \ start \\ & & \mathtt{++} \ buf \ \mathtt{to} \ total \\ & & \mathtt{++} \ \mathtt{concat}(content_0) \ \mathtt{from} \ (start + total) \end{array} \tag{8.14}$$

The key idea behind the proofs to propagate the invariant through the loop is to normalize all terms of type stream to a representation with ++. For example, the effect of `afs_writepage` is captured by the equality

$$\begin{aligned}&\texttt{concat}(\mathit{content}[\mathit{pageno} \mapsto \mathit{page}])\\ &\quad = \quad \texttt{concat}(\mathit{content})\ \texttt{to}\ (\mathit{pageno} \cdot \texttt{VFS\_PAGE\_SIZE})\\ &\qquad \texttt{++}\ \mathit{page}\\ &\qquad \texttt{++}\ \texttt{concat}(\mathit{content})\ \texttt{from}\ (\mathit{pageno} \cdot (\texttt{VFS\_PAGE\_SIZE} + 1))\end{aligned}$$

A similar theorem exists for $\texttt{copy}(\mathit{buf}, \mathit{total}, \mathit{offset}, n; \mathit{page})$. Equation (8.14) is then restored by distribution lemmas such as $(\sigma_1 \texttt{++} \sigma_2)\ \texttt{from}\ n = \sigma_2\ \texttt{from}\ (n - \#\sigma_1)$ if $n \geq \#\sigma_1$, and by cancellation of leading stream components of both sides of the equation $(\sigma \texttt{++} \sigma_1 = \sigma \texttt{++} \sigma_2) \leftrightarrow \sigma_1 = \sigma_2$ for finite σ. Finally, the loop invariant is mapped to the respective abstract POSIX operation.

Compared to the canonical alternative—a formulation of the loop invariants with `splice` (resp. `copy` for reading)—our approach is considerably more elegant: Invariant (8.14) does not need to mention the "current" size of the file, which would lead to case distinctions whether the file needs to grow. Such case distinctions (also found in `max`) produce a quadratic number of cases in the proof as one needs to consider the previous *and* the new version of the invariant.

8.9 Related Work

The model by Hesselink and Lali [81] treats files as atomic entities, focusing entirely on the tree structure.

The model by Damchoom and Butler [39] breaks files down to the level of pages and maps these directly to the physical structure of the flash hardware. The work is motivated from a more theoretical point of view, namely as a demonstration of event decomposition in the Event-B framework [6]. The approach taken is somewhat unrealistic: it is assumed that the pages of the hardware can be programmed individually, atomically, and in parallel. All of these assumptions do not hold for modern flash hardware (cf. Section 2.3.1 and Chapter 10).

The study by Arkoudas et al. [10] seems to be the earliest effort to formally model and prove properties of page-based reading and writing. While the underlying data structure is realistic (essentially what has been presented in Chapter 8), the *interface* presented to the outside world permits a client to access single bytes at a time only, which is clearly not satisfactory in practice.

Reading and writing files at byte-level has been addressed also by Kang and Jackson [92]. They have built a vertical prototype in the specification language Alloy[2] that breaks the linear view of files down to the pages of the flash hardware (similarly to [39]). They validate an algorithm corresponding to the `read` and `write` implementation of VFS (Section 8.6) for small bounded state spaces. However, their algorithm relies on an explicitly precomputed list of subranges of the pages to read/write in order to accommodate the limitations of the model checker. In Flashix, page access is determined on the fly without the need for an intermediate data structure.

[2] http://alloy.mit.edu/

Another approach is taken for the verified file system FSCQ [32, 34], the abstract specification in [65], and the elaborate SibylFS [132]: the directory tree is modeled as a pointer structure with *forward* links, thereby supporting hard-links, although these are not realized in FSCQ. This design skips the first refinement step of Flashix in favor of starting with a more concrete specification in the first place. The approach can be justified well. In both [34, 65], separation logic is used, which permits to reason about pointer structures effectively as if they were algebraic trees, whereas the model in [132] serves as a *test oracle* where complexity of the specification is not so much an issue.

SibylFS is a conformance checker for file systems by Ridge et al. [132]. The goal of the project is to provide a highly precise formal model of the POSIX file system interface, which can serve as a test oracle. The model is written in Lem [111], a language bridging functional programming and formal logic. The authors have taken care to model many subtleties of POSIX implementations, including deviations from the standard in Linux or Mac OS X. The model is vast in comparison to the one presented here (several thousand lines of code), and although it is very suitable for the intended task, it may be much too complex to serve as a high-level specification for a formal verification.

FSCQ [34] is a complete implementation and produces running code at the end (based on the FUSE library like Flashix) It thereby includes much of the complexities of the VFS (for instance reading and writing).

In her recent thesis, Najafzadeh [113] describes a formal graph-based POSIX model similar ours VFS/AFS. The models serves as a case study for weak or "hybrid" consistency invariants in the presence of concurrency and replication of state. These invariants are classified and uniform approaches to proving their preservation are presented. The analysis is based on partial and total causal orderings between operations, and their commutations and stability in a concurrent context. For the POSIX case study, it is shown that the data structure representing the file system remains a proper tree.

The work is backed by a tool CISE [114] ("Cause I'm Strong Enough"), which automatically checks for such properties. According to Najafzadeh [113, Section 8 and 9] the tool helped to detect and remove unneeded synchronization between operations in many cases and also uncovered that moving files around is not safe without such synchronization. The properties considered in this work are beyond the scope of this thesis, and it is not clear, how the analysis would integrate with refinement. Nevertheless, the ideas presented are interesting for the future development of Flashix. The locking mechanism is based on the specification of conflict relations $T_1 \bowtie T_2$ between two sets of tokens that link concurrent operations (akin to the conflict relation $\#$ of event structures [118]).

Chapter 9

Flash File System Internals

Efficiently Dealing with Flash Memory Characteristics

Summary. This chapter continues the development of the formal models of Flashix in the refinement hierarchy from the core concepts of a flash file system implementation. It implements efficient strategies to store data on flash memory, while at the same time the concepts are introduced as abstractly as possible. Central to the flash file system is the life cycle of on-flash data that is always written to fresh locations by operations.

Publications: This chapter is based on [54] and the work of Schierl et al. [147].

Contents

This section presents the formal model of the core concepts of the Flashix file system, denoted by FFS or "core" in the following. It is based on work by Schierl et al. [147] pioneering the early stages of the project, while additional aspects and the modularization in terms the the index and the journal are described as part of [54].

The file system core implements the AFS interface from the previous section. It introduces the first artifacts that are specifically tailored towards flash memory. Recall that storage is structured into erase blocks that themselves are partitioned into pages. Write access is provided at the granularity of whole pages. Modern hardware typically supports writing pages within a block in sequential order only. To reuse storage space, blocks can be erased in their entirety. Reading is supported for random subranges of a block.[1]

Although these details are abstracted away here as much as possible, they nevertheless determine the fundamental approach taken in the file system core: It stores a unordered collection of data items, called "nodes", which represent files, directories, directory entries, and pages that store the content of files and correspond directly to their counterparts of the VFS data model described in Section 8.1. New versions of nodes are always written

[1] At least, if the hardware doesn't support unaligned reads, the driver interface can easily emulate such a behavior. With Linux MTD this can be relied on.

to fresh locations on the storage medium as in-place updates are impossible. Since data moves around all the time, yet another indirection via an index is required that tracks the addresses of the most recent copy of a given node. The strategy has already been sketched with Example 5.5.

Management of on-flash storage space is delegated to the *journal* subcomponent, which also provides the necessary means to group multiple write operations into transactions that take effect atomically wrt. power cuts. Likewise, the *index* is also realized in its own submodule, implemented as a B^+ tree. The core combines the functionality of the journal and the index to implement an AFS-compatible file system that is linked into the VFS in the generated code. This means that while the two subcomponents are refined further, the glue code presented here that ties them together is part of the final system.

Remark. Actually, it is not possible to strictly separate the index and the journal into their own isolated submachines. For instance, the journal must be able to determine which addresses are still in use by the index (for garbage collection), some operations (e.g., commit and recovery) cannot be specified separately, because they modify data structures of both layers at the same time. There is also a number of invariants that ties the layers closely together. For this reason, the index and the journal are realized as *one* machine in the KIV mechanization (that is refined as a whole). We keep the presentation separate here when it makes sense but take a combined view when it becomes necessary.

9.1 General Strategy

The state of the index component is given by the RAM index *ri* and the flash index *fi*. All operations except commit and recovery access the RAM index only. The abstract state of the journal is given by an unordered flash store *fs*. The *log* records the addresses of those nodes that have been written recently, for which the RAM and flash address differ.

The high-level flow of data as affected by external and internal operations of the flash file system core is visualized in Figure 9.1. The state is partitioned into a volatile part *ri* and a persistent part *fi*, *fs*, and *log* (attribution at the top) managed by the index and the journal (attribution at the bottom). White boxes represent the constituents of the state. Arrows indicate which part of the state flows through which kind of operation. These are denoted by grey boxes and encompass 1. regular operations from the AFS interface, and several internal ones, namely 2. recovery and 3. commit as well as 4. garbage collection of on flash storage space. It is assumed that initially the RAM index and the on-flash version coincide ($ri = fi$), there is some file content *fs* present and also some free space (indicated by $\varnothing$), and the log is empty ($[\,]$).

1. Regular operations operate on the RAM index and write new entries to the flash store. The flash index is not modified immediately, instead the difference between the two indices is recorded in the log. The part *fs* remains unchanged but some part potentially becomes obsolete due to out-of-place updates replacing previous versions of data. In the figure, this is indicated by splitting *fs* into *old* and fs'.
2. The recovery operation takes over after a power cut to restore the RAM index, which resides in volatile memory and is therefore lost, from the persistent data structures. Refering to the figure, $ri + new$ is reconstructed from $fi (= ri)$ and the *new* part from the log and the flash store. Recovery is the only operation that reads the on-flash index *fi*.

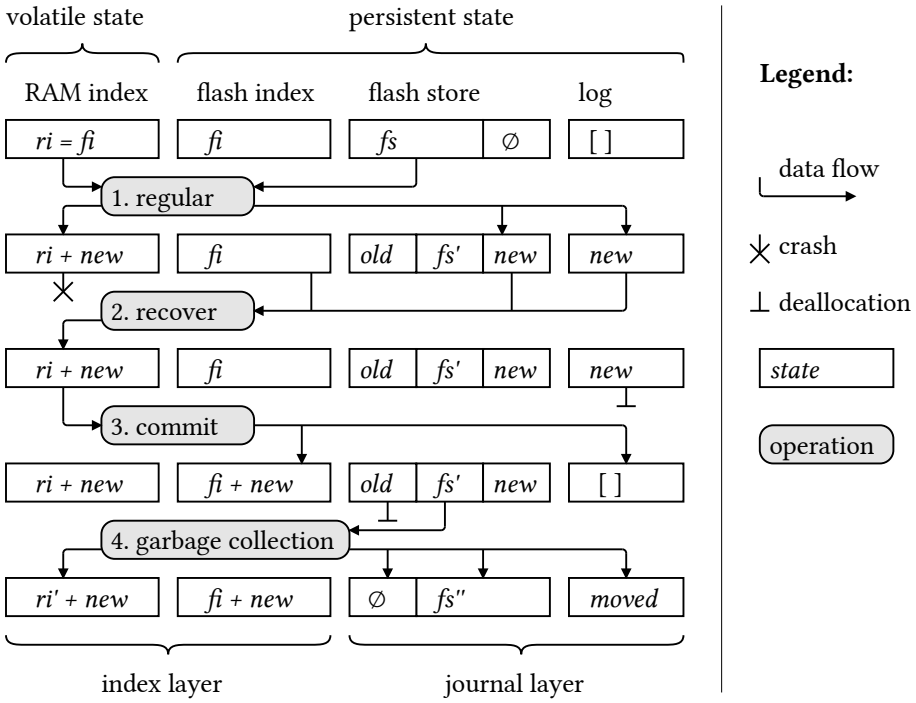


Figure 9.1: Data life cycle and effect of operations on the state (illustration adapted from Schierl et al. [147, Figure 1]).

3. Purpose of the commit operation is to re-establish the initial situation: the flash index is updated to reflect the in-memory version and thus the log can be emptied. Commit is the only operation that writes the on-flash index *fi*.
4. Garbage collection clears up the *old* part in the flash store, potentially moving some live data around that happens to sit in almost-empty blocks (updating *ri* as well). Like regular operations, garbage collection modifies the RAM index, the log and the flash store, but not the flash index.

The integration of the data structures of the index and the journal is shown in Figure 9.2, complementing the previous views in Figure 7.1 and Figure 8.1, as well as the dynamics in Figure 9.1. At the bottom the flash memory is visualized as an unstructured storage (by its cloudy shape). At the top, the index is shown: a current version (grey triangle) in main memory encompasses all modifications, and the outdated flash version flash (black triangle) shares some unchanged part (white) with the RAM index. The log (at the bottom) encodes the changes that transform the outdated flash index to the current one in main memory.

The example state in Figure 9.2 demonstrates the links between the index, flash store, and log for new objects generated by creating a file named `thesis.tex` (Example 6.2). The updated parent directory (folder symbol), the encoded directory entry carrying the name of the file (denoted by the big arrow in the middle) and the encoded inode object storing the metadata of the new file (visualized as a leaf of paper) are part of the file store and are referenced by the latest three log entries. The index is also updated to point to the new objects stored on flash, in Figure 9.2 these updates affect the grey part of the index.

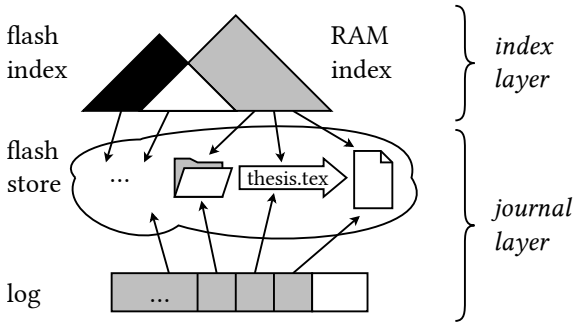


Figure 9.2: Conceptual view of the flash store, the index, and the log, showing the directory inode (folder symbol), the directory entry labeled "thesis.tex" and the inode of the new file (leaf of paper).

9.2 Specification of the Journal and the Index

The state consisting of the two indices, the flash store, and the log is specified abstractly by algebraic maps resp. a list:

$$\textbf{spec vars}\ \text{(index)} \quad ri, fi : \mathit{Key} \rightarrow\!\!\!\!\!\!+\ \mathit{Address}$$

$$\textbf{spec vars}\ \text{(journal)} \quad fs : \mathit{Address} \rightarrow\!\!\!\!\!\!+\ \mathit{Node},\ log : \mathit{List}\langle \mathit{Address} \rangle$$

The objects representing keys and nodes are defined by algebraic data types with constructors resembling the data model of the VFS shown in Section 8.1. The idea is that whenever a key is present in the index, then a node of the corresponding type is stored on flash at that address.

$$\begin{aligned}
\textbf{data}\ \mathit{Key} = {} & \texttt{inodekey}(\texttt{ino}: \mathit{Ino}) \\
\mid {} & \texttt{dentrykey}(\texttt{ino}: \mathit{Ino},\ \texttt{name}: \mathit{String}) \\
\mid {} & \texttt{datakey}(\texttt{ino}: \mathit{Ino},\ \texttt{pageno}: \mathit{Ino}) \\
\textbf{data}\ \mathit{Node} = {} & \texttt{inodenode}(\texttt{key}: \mathit{Key},\ \texttt{meta}: \mathit{MetaData},\ \texttt{dir?}: \mathit{Bool}, \\
& \qquad \texttt{nlink}: \mathit{Ino},\ \texttt{size}: \mathit{Ino},\ \texttt{nsubdirs}: \mathit{Ino}) \\
\mid {} & \texttt{dentrynode}(\texttt{key}: \mathit{Key},\ \texttt{target}: \mathit{Ino}) \\
\mid {} & \texttt{datanode}(\texttt{key}: \mathit{Key},\ \texttt{data}: \mathit{Page})
\end{aligned}$$

Inodes are uniquely determined by their numbers. They store meta data `meta`, whether they belong to a directory, the number of hard links (equal to 1 for directories except the root), the size of a file resp. the number of directory entries, of which `nsubdirs` determines how many of these are again directories in the latter case. Directory entries are uniquely identified by the inode number of the parent directory and a name. The corresponding node references the inode number of the child, i.e., the target of the edge in the graph structure of the directory hierarchy. The data pages making up the file's content are uniquely identified by the file they belong to and the sequential number of the page within the file. The node simply stores the contained data as an array of bytes with the length `VFS_PAGE_SIZE`.

In addition, each node stores the key which references it in the index to permit reverse lookup. For instance, it is possible to check whether a node read from flash is still refer-

enced by the index by $fs[adr].\texttt{key} \in \texttt{dom}(ri)$, which is relevant for garbage collection (see Section 9.5). Reverse lookup is crucial for for power cut recovery, too (see Section 12.3).

The journal has an operation to read a node from flash, and operations to store groups of several nodes $nd_{1\cdots n}$, extending the *log* at the same time.

```
jnl_get(adr;  nd, err)
  precondition adr ∈ fs
    { nd := fs[adr], err := ESUCCESS }
  or { choose err ∈ {EIO, ···} }
```

```
jnl_append_n(nd_1, ···, nd_n;  adr_1, ···, adr_n,  err)
  precondition true
    { choose adr_{1···n} ∉ dom(fs) distinct
      fs  := fs[adr_1 ↦ nd_1, ···, adr_n ↦ nd_n]
      log := log ++ [adr_1, ···, adr_n]
      err := ESUCCESS }
  or { choose err ∈ {EIO, ···} }
```

The operation `jnl_get` just returns the node *nd* associated with the address *adr*. The operation `jnl_append` takes *n* nodes, stores these on disk and returns their locations on flash memory $adr_{1\cdots n}$, which are later stored in the index. It encodes the restriction that no data can be overwritten on flash: the addresses where the nodes are stored are chosen to be fresh and not already part of *fs*. Furthermore, the intended interpretation is that the assignment to *fs* and *log* happen atomically and at the same time. This encodes that an implementation must be able to achieve the effect observably atomic with respect to power cuts.

Like we have seen previously in Section 8.3, the journal exhibits nondeterministic errors as it accesses the flash memory. The index operations to lookup, store, and remove mappings directly refer to their algebraic counterparts (left column).

```
idx_lookup(key; adr, exists)
  exists := (key ∈ ri)
  if exists then adr := ri[key]

idx_store(key, adr)
  precondition adr ∈ fs ∧ adr ∉ ran(ri)
  ri[key] := adr

idx_remove(key)
  precondition key ∈ dom(ri)
  ri := ri − key
```

```
idx_dentries(key; keys)
  keys := entries(key)

idx_trunc(key, n)
  ri := ri \ data_≥(key, n)

idx_newino(; key)
  choose ino
  with ino ≠ 0 ∧ inodekey(ino) ∉ ri
  in key := inodekey(ino)
```

Looking up a key returns whether it is part of the index and provides the respective address in that case. The precondition $adr \in fs$ for `idx_store` guards the index implementation from having to deal with unallocated addresses, leading to the formal invariant (9.5) below. The non-obvious precondition $adr \notin \texttt{ran}(ri)$ for `idx_store` prevents the same address from being used twice. Technically, it means that the index is an injective map, which is exploited to incrementally track the addresses referenced by *ri*: From $key \in \texttt{dom}(ri)$ for `idx_delete` one can conclude that the address stored at $ri[key]$ becomes obsolete.

Additional operations include reading the entries of a directory, removing all data keys for pages above a given file size *n*, and selecting an unused inode number that can be used to identify a new file or directory. The algebraic helper `entries` selects the directory entries with the inode number matching the directory given by *key*, whereas $\texttt{data}_{\geq}$ selects those

data keys that become obsolete by restricting the file size for *key* to *n*. This corresponds to the `upto` function (8.2) of the abstract level.

$$\begin{aligned} \texttt{entries}(key) &= \{\texttt{dentrykey}(ino, name) \in ri \mid ino = key.\texttt{ino}\} \\ \texttt{data}_{\geq}(key, n) &= \{\texttt{datakey}(ino, pageno) \in ri \mid ino = key.\texttt{ino} \\ &\qquad \wedge pageno \cdot \texttt{VFS\_PAGE\_SIZE} \geq n\} \end{aligned} \tag{9.1}$$

9.3 Regular Operations

With the prerequisites on the two subcomponents, it is now time to show how these are integrated by the flash file system in order to realize the AFS interface.

Figure 9.3 shows as an example the regular operations `fs_lookup` and `fs_create` which realize the AFS specification from Figure 8.4.

The lookup operation receives the parent's inode number and a file name (as part of *dent*). The precondition ensures that the parent is a directory. Via the index it is determined whether there is an entry with that name, i.e., the specified `dentrykey` is contained in *ri*. If so, the address *adr* holds the location of the corresponding node on disk, which can be retrieved via the journal (effectively $nd = fs[ri[\texttt{dentrykey}(pino, name)]]$). The return value is determined in *dent* just as in the specification, except when reading from the flash storage failed ($err \neq \texttt{ESUCCESS}$ after `jnl_get`), in which case this error is passed on to the caller of `fs_create` with an unmodified *dent*.

The create operation receives the VFS inode object of the parent directory, and the name (as part of *dent*) and the metadata *md* of the new file. A fresh inode number $fino : \mathit{Ino}$ for the new file is requested from the index by `idx_newino`. Several keys and nodes are prepared: The parent directory identified by *pino* is affected because its size changes (all other fields are kept unchanged from the *inode*). The key for the directory entry links the file under *pino* with the given *name*, and the corresponding node stores the target *fino*. The file itself is created with the provided metadata, the directory flag set to `false`, a link count of 1 and a size of zero (and a dummy value 0 for the number of subdirectories). The nodes are written to disk via the journal layer, which yields three addresses as outputs. The index is updated at the end, but only if the write to flash was successful ($err = \texttt{ESUCCESS}$).

Most other operations of the AFS interface Chapter 8 are realized by this pattern (cf. the first row in Figure 9.1): preparing a few keys and nodes, writing these to disk, and finally updating the index.

Some operations, such as `unlink` or `rmdir` delete keys from the index instead of storing new mappings. In order to record such deletions as durable on flash the affected inode and dentry nodes are written with an `nlink` resp. `target` of zero. This is shown in Figure 9.4, which removes a hard-link to a file. Like for `create`, three keys and nodes are prepared. The second last line in the listing removes the key from the RAM index, which is does not have a durable effect. In the event of a power cut, the effect of the unlink operation would be lost and even worse, the `size` of the parent inode would be wrong. For this purpose a directory entry node containing a zero target inode number is written, which signifies its deletion in a persistent way. Note that the file itself is removed only when the link count drops to zero and no further open file handles are pointing to it by the `evict` operation (not shown for the flash file system core), which is called by the VFS as described in Section 8.4.

```
fs_lookup(pino; dent, err)
  let key = inodekey(pino)
      name = dent.name in
  precondition key ∈ dom(ri) ∧ fs[ri[key]].dir?
  idx_lookup(dentrykey(pino, name); adr, exists)
  if exists then
     jnl_get(adr; nd, err)
     if err = ESUCCESS then dent := dentry(name, nd.target)
  else
     err := ENOENT, dent := negdentry(name)


fs_create(md; inode, dent, err)
  let pino = inode.ino
      name = dent.name in
  precondition   inodekey(pino) ∈ dom(ri)
                 ∧ fs[ri[inodekey(pino)]].dir?
                 ∧ dentrykey(pino, name) ∉ dom(ri)
  idx_newino(; fino)
  key_1 := inodekey(pino)          nd_1 := inodenode(key_1, ···, inode.size + 1, ···)
  key_2 := dentrykey(pino, name)   nd_2 := dentrynode(key_2, fino)
  key_3 := inodekey(fino)          nd_3 := inodenode(key_3, md, false, 1, 0, 0)
  jnl_append_3(nd_1, nd_2, nd_3;  adr_1, adr_2, adr_3, err)
  if err = ESUCCESS then
    idx_store(key_1, adr_1)
    idx_store(key_2, adr_2)
    idx_store(key_3, adr_3)
```

Figure 9.3: File system core: implementation of the operations `lookup` and `create`. (The **`let`**s are moved to the beginning to shorten the presentation).

```
fs_unlink(md; inode, dent, err)
  let pino = inode.ino
      name = dent.name in
  precondition   inodekey(pino) ∈ dom(ri)
                 ∧ fs[ri[inodekey(pino)]].dir?
                 ∧ dentrykey(pino, name) ∈ dom(ri)
  idx_newino(; fino)
  key_1 := inodekey(pino)          nd_1 := inodenode(key_1, ···, inode.size − 1, ···)
  key_2 := dentrykey(pino, name)   nd_2 := dentrynode(key_2, 0)
  key_3 := inodekey(fino)          nd_3 := inodenode(key_3, ···, inode.nlink − 1, ···)
  jnl_append_3(nd_1, nd_2, nd_3;  adr_1, adr_2, adr_3, err)
  if err = ESUCCESS then
    idx_store(key_1, adr_1)
    idx_remove(key_2)
    idx_store(key_3, adr_3)
```

Figure 9.4: File system core: implementation of operation `unlink` to remove a hard-link to a given file. In contrast to creation, the node encoding the directory entry contains a zero target inode, which signifies its deletion. The corresponding key is deleted from the index in the second last line.

9.4 Commit and Recovery

This section briefly describes the commit and recovery operations without going into much of the details, which will be discussed in the presence of crashes in Section 12.3.

The commit operation is shown below. It copies the RAM index to flash and clears the log. Since commit writes to flash memory it exposes potential hardware errors.

```
journal+index_commit(; err)
    { fi := ri, log := [], err := ESUCCESS }
  or { choose err ∈ {EIO, ···} }
```

The RAM index determines exactly, which part of the flash memory constitutes the observable file system state, more precisely, its composition $fs \circ ri$ with the flash store. However, in the event of a power cut the RAM state is lost. That the RAM index is truly redundant and can be recovered to its previous state after a power cut is expressed by

$$\textbf{invariant}\ (\text{FFS}) \quad ri = \texttt{replay}(log, fi, fs), \tag{9.2}$$

where `replay` is part of the system initialization resp. the recovery procedure that is called after a power cut. The function traverses the *log* from oldest to newest and (re-)applies all missing operations to the outdated *fi*. The details of this recovery process are postponed to Chapter 12, for now it suffices to consider the *existence* of this connection to warrant that the RAM index is functionally dependent on the persistent state. Note that after a commit, the invariant trivially holds, because $\texttt{replay}([\,], fi, fs) := fi$ for the empty log, which equals *ri*. In the implementation of the flash file system core, the effect of the `replay` function is realized programmatically.

9.5 Garbage Collection

As operations such as `fs_create` shown in Figure 9.3 store new data at new addresses in *fs* but never delete something, it becomes necessary to periodically cleanup *fs* by a garbage collection procedure. Specifically, addresses $adr \notin \texttt{ran}(ri)$ point to obsolete nodes belonging to files, directories, entries and data that have been modified so that a new version exists elsewhere, or that have been deleted entirely.

Intuitively, an abstract specification of garbage collection omits a couple of such addresses by repetition of transitions

$$fs' = fs \setminus \{adr\} \quad \text{where} \quad adr \notin \texttt{ran}(ri) \wedge adr \notin \texttt{dom}(log), \tag{9.3}$$

where the domain of the list $log\colon \texttt{List}\langle\texttt{Address}\rangle$ is simply the set of its elements. However, the matter is not as simple.

First, the index is accessible (efficiently) only by keys. Each node *nd* stores its respective key, denoted by $nd.\texttt{key}$, and thus one can equivalently check $fs[adr].\texttt{key} \notin \texttt{dom}(ri)$. The required invariant is (9.6) (see below).

Second, as data stored in blocks that can be reclaimed in their entirety only, it may be necessary to move some nodes out of an almost-empty block before erasing it. These moves are represented by transitions of the form

$$\begin{aligned} fs' &= fs[adr \mapsto fs[ri[key]]] \quad \text{for} \quad key \in ri \text{ and } adr \notin \texttt{dom}(fs) \\ log' &= log \mathbin{+\!\!+} [adr] \\ ri' &= ri[key \mapsto adr'] \end{aligned} \tag{9.4}$$

that update both the index and the log also and are intermixed with deletion steps (9.3).

Correctness of garbage collection requires one to prove that it is not observable from the outside, i.e., it corresponds to a non-operation when abstracted to AFS. It can easily be checked that (9.3) and (9.4) both preserve $fs \circ ri$. However, note that a garbage collection step may invalidate addresses contained in the outdated on-flash version of the index, i.e., $adr \notin \texttt{ran}(ri)$ and $adr \notin \texttt{dom}(log)$ do not imply that $adr \notin \texttt{ran}(fi)$, which will become relevant when power cuts are considered, because then fi is read.

9.6 Invariants

The index and the journal are tightly integrated. For example, the garbage collection algorithm implemented as part of the journal needs to refer to the index in order to determine whether a node is still in use. The tight integration also manifests in invariants over the combined state space and preconditions that refer to both data structures. For that reason we present the two layers at the same time (and in the development in KIV these are combined into one ASM as remarked previously).

The following invariants can already be maintained internally to the index and journal without relying on the concrete mappings stored by the flash file system core. They are proved easily from the preconditions of the operations.

invariants (index & journal) (9.5)

$\texttt{ran}(ri) \subseteq \texttt{dom}(fs)$

$\texttt{injective}(ri)$

$\texttt{dom}(log) \subseteq dom(fs)$

The flash file system core maintains some further invariants in addition to (9.5) that describe well-formed states. The main purpose is to connect the keys stored in the index to the nodes on stored flash. The following invariant ensures that the keys stored in the nodes are the correct ones. It is used for reverse lookups when only an address is known during garbage collection as described already and during recovery, which reads such addresses from the log.

invariant (FFS) $\quad \forall\, key \in \texttt{dom}(ri).\ fs[ri[key]].\texttt{key} = key$ (9.6)

Many invariants have been described in [147] already. For instance,

- The types of each key must match the type of the corresponding node, e.g., a key for an inode should never lead to a directory entry
- As an equivalent to (8.6), the target of a directory entry node is a valid inode number ino such that $\texttt{inodekey}(ino) \in \texttt{dom}(ri)$. This ensures that path lookup doesn't get stuck.
- The parent of a dentry key is valid—this is an invariant that was implicit in the AFS model, whereas here it must be stated explicitly. Purpose of this invariant is to prevent superfluous pages floating around in the index, which could otherwise suddenly be associated with a newly created directory. Similarly, the ino referenced by data keys must denote a falid file.

The key point to these invariants is that a valid representation of the state by fs and ri is less constrained than the *dirs* and *files* of AFS by construction and must therefore be narrowed down explicitly.

9.7 Verification

This section sketches the key steps in the verification of the Flash File System wrt. its specification, the AFS model from Chapter 8, i.e., we prove FFS(index+journal) $\sqsubseteq$ AFS.

9.7.1 Abstraction to AFS

The data structures found in the flash file system core are mapped to the AFS *dirs* and *files* by folding two layers of indirection, namely, by collapsing the intermediate addresses that connect the RAM index *ri* to the flash store *fs* and by nesting the nodes for directory entries and data pages into the stores of the respective directories and files. The flash index *fi* and the *log*, however, are irrelevant for the abstraction here, since they come into play only in the event of a power cut and subsequent recover. This issue is addressed in Chapter 12 by an extension of the AFS model that considers the distinction into RAM and flash state explicitly.

The abstraction to the AFS data structures is presented in terms of the extensional properties of the two stores $dirs = \texttt{dirs}(ri, fs)$ and $files = \texttt{files}(ri, fs)$ in terms *ri* and *fs* as follows.

coupling (AFS $\sqsubseteq$ FFS(index+journal))

$$\texttt{dom}(dirs) = \{ino \mid \texttt{inodekey}(ino) \in ri \wedge fs[ri[\texttt{inodekey}(ino)]].\texttt{dir}?\}$$
$$\texttt{dom}(files) = \{ino \mid \texttt{inodekey}(ino) \in ri \wedge \neg\, fs[ri[\texttt{inodekey}(ino)]].\texttt{dir}?\}$$

For a (parent) directory $pino \in \texttt{dom}(dirs)$ and $nd = fs[ri[\texttt{inodekey}(pino)]]$ it is required that

coupling

$$dirs[pino] = \texttt{dir}(nd.\texttt{meta}, nd.\texttt{size}, nd.\texttt{nlink}, nd.\texttt{nsubdirs}, entries) \quad \text{such that}$$
$$entries = [name \mapsto ino \mid \texttt{dentrykey}(pino, name) \in ri \wedge ino = fs[ri[\texttt{dentrykey}(pino, name)].\texttt{target}],$$

i.e., the metadata attributes of the directory are taken from the node on flash, whereas the map *entries* collects all the directory entries in terms of the keys with the right parent inode number *pino*. The target *ino* of such an entry must be read from flash.

A file $fino \in \texttt{dom}(files)$ is assembled similarly for $nd = fs[ri[\texttt{inodekey}(fino)]]$ by

coupling

$$files[fino] = \texttt{file}(nd.\texttt{meta}, nd.\texttt{size}, nd.\texttt{nlink}, content) \quad \text{such that}$$
$$content = [n \mapsto page \mid \texttt{datakey}(fino, n) \in ri \wedge page = fs[ri[\texttt{dentrykey}(fino, n)].\texttt{data}],$$

such that the content consists of all the pages with a corresponding data node on flash.

Note that this abstraction is only well-defined if the FFS core invariants (9.5) hold to guarantee that for a *key* the lookups $fs[ri[key]]$ fall are inside domains of *ri* resp. *fi*. The abstraction guarantees for example, that *dirs* and *files* are disjoint. Furthermore, the abstraction is functional, i.e., the AFS state is uniquely determined by its FFS counterpart.

9.7.2 Refinement Proofs

As we have seen previously in Section 8.8.2, symbolic execution of the abstract and concrete operation produces sequents of the form

$$\Gamma \vdash \mathtt{R}(dirs', files', ri', fs') \wedge out_1 = out_2$$

where $\Gamma = \mathtt{R}(dirs, files, ri, fs), \cdots$ contains the initial instance of the simulation relation, as well as preconditions and other information that has been gathered during symbolic execution (e.g., results of the tests in conditionals and subroutine postconditions). Concretely, the proof goal will contain instances of

$$\Gamma \vdash files' = \mathtt{files}(ri', fs') \wedge dirs' = \mathtt{dirs}(ri', fs'),$$

which is solved by applying the standard extensionality principle of finite maps:

$$m_1 = m_2 \;\leftrightarrow\; \mathtt{dom}(m_1) = \mathtt{dom}(m_2) \wedge \forall\, k \in \mathtt{dom}(m_1).\, m_1[k] = m_2[m].$$

It states that two maps are equal exactly when their domains coincide and when lookup of a given key k in the domains produces the same result. This breaks down the proofs to a stage where the definitions of the coupling relations in Section 9.7.1 apply. It turns out that simply unfolding these definitions suffices in almost all cases to close the goal by discerning the case distinctions whether a given inode number $ino \in \mathtt{dom}(dirs')$ resp. $ino \in \mathtt{dom}(files')$ has been modified by the operation.

The second stage of the proof is concerned with comparing the store of directory entries *entries* resp. the *content* of files, which is also done by extensionality. Only the `truncate` operation leads to a complex path in the proof, which compares the algebraic function *content* `upto` n from (8.2) to its counterpart $ri \setminus \mathtt{data}_{\geq}(key, n)$ from (9.1). There is no conceptual difficulty other than tedious case distinctions that can be automated to a reasonable degree.

The initial approach to prove this refinement involved a high number of lemmas to propagate modifications through the abstraction relation, a simple example is

$$\mathtt{dirs}(ri - \mathtt{inodekey}(ino), fs) = \mathtt{dirs}(ri, fs) - ino.$$

Unfortunately, most of these commutations, in particular when directory entries or pages are accessed, require preconditions from the invariants of the flash file system (see Section 9.6. These are tricky or impossible to establish for partially modified states ri' as in

$$\mathtt{dirs}(ri'[\mathtt{dentrykey}(ino, name), adr], fs),$$

when ri' contains some other modifications in comparison to the original ri.

9.8 Related Work

The initial prototype model of the core concepts of UBIFS by Schierl et al. [147] has served as the anchor for the whole development. The model presented here follows the same general approach and is realized similarly by a characterization of the state in terms of ri, fi, log, fs

A number of extensions have been introduced as well: to accommodate nondeterministic hardware errors, all operations that access the flash storage can now fail. Deletion has been factored into two operations, unlink and evict, and file handles that were previously not stored at all are now recorded by the VFS (see Chapter 8). As a consequence, orphaned files (cf. Section 7.6) are now handled by the model, although the presentation of this is postponed to Section 12.3. As a technical aspect, the invariants have been restated for better proof automation. The refinement proof connecting the flash file system model to AFS is entirely new.

The work at Data 61 (formerly known as NICTA) follows a subset of the goals of the Flashix project. They aim at a fully verified file system for flash memory but only realize the part of the refinement chain from the AFS specification downwards to the erase block management layer. In practice, their file system BilbyFS introduced in [93] runs in the Linux kernel and can therefore take advantage of the existing Virtual Filesystem Switch as well as the UBI interface.

Research at Data 61 focuses specifically on the construction of verified C code, which brings in many low-level aspects that are out of the scope of this thesis. The approach taken is to use a domain specific language called COGENT [9] designed for file system implementations (previously described as CDSL in the technical report [121]). Besides BilbyFS they have written an Ext2-like file system in this language as well and demonstrate in [9] that this approach leads to efficient code on one hand and to an automatic abstraction to a functional model of the C Code into the Isabelle system [119]. Closer inspection of the COGENT code reveals that it is similar to the flash file system core in this chapter, in particular the cascading chains of error handling are present there as well.

However, the design of BilbyFS is deliberately kept much simpler than UBIFS and Flashix: The index is not stored on flash memory and must be reconstructed at every boot just like the tables storing information about free space.

The file system FSCQ [32–34] is a file system for conventional magnetic storage inspired by the design of the relatively simple `xv6` file system [37]. It is fully verified with the help of the theorem prover Coq [20] in a logic called Crash Hoare Logic, which permits them to prove power cut safety. Running Haskell code is automatically derived from the specifications by Coq's extraction mechanism. It is integrated into Linux using the FUSE library like Flashix (see Section 2.4.1).

FSCQ operates on standard block devices exposed by the Linux kernel. These have a similar structure as flash memory blocks but without the restrictions inherent to the latter. FSCQ exploits this (like many conventional file systems) to take the blocks as basic primitives. For example, the content of a directory is stored in a linked list of several of these blocks, where it is possible to update individual entries by sub-block writes.

Internal to FSCQ is a component that implements a sequential log maintaining a list of pending blocks to be written to disk. In principle, this can be done asynchronously, although for reasons fo crash-safety, the log is flushed with each POSIX operation. The log in FSCQ resembles much the erase block management layer, contrasting the transactional journal in the Flashix core (Section 9.2), because the latter operates on smaller data quantities (the nodes), whereas in FSCQ the upper layers are already responsible to encode data down to byte-based blocks.

The log is backed by a buffer cache that speeds up read operations. An interesting aspect is that they integrate unverified code for the cache-replacement strategy. The actions determined by the replacement algorithm are instead validated to conform to certain

safety properties by a verified check.

There is also not much need for an index in FSCQ, as blocks are allocated directly to the files and directories and this mapping does not need to change later on.

The verification of FSCQ also includes the serialization and deserialization between in-memory data structures and their byte representation. In Flashix, such routines are currently outside of the scope of these and are instead synthesized by the code generator.

In a recent master's thesis, Wang [155] has developed an improved version called Rapid FSCQ, which considers checksum-based logging and a verified implementation of hash-algorithms for caching to speed up the file system's operations. She has shown that this file system performs comparably to Ext4.

Chen's thesis [32] extends FSCQ by asynchronous writes and a specification of the POSIX operations `fsync` and `fdatasync`. This permits to implement and verify *write-back* caching. It is specified by recording the entire history of the file system. Out-of-order writes may rewrite this history. Such caching is beyond the scope of this thesis and will be explored in the future.

Galloway et al. [63] abstract the existing Linux VFS code to a SPIN model to check correct usage of locks and reference counters. With a similar approach, Taverne and Pronk [151] consider a hand-crafted abstract model of a flash file system called RAFFS amenable to model checking, however, the model does not incorporate much details of flash FS internals due to a limited search space. Work that directly checks the C source code of existing file systems is by Mühlberg and Lüttgen [112], Yang et al. [161]. These approaches limit themselves to specific properties that are weaker than functional correctness (e.g., memory safety) or cover concepts orthogonal to this work (e.g., correct usage of locks). Nevertheless, they are an impressive achievement, notably because they have uncovered real bugs in code that is used on a daily basis.

The Linux Verification Center[2] takes these efforts to a holistic scope: they routinely check large parts of the Linux kernel and contribute a continuous stream of bug-reports.

It should be noted that these automated techniques complement the approach taken in this thesis. Full functional verification of *existing* code is a rather tedious undertaking (see e.g. Baumann et al. [16, 17]), it can be worthwhile in some circumstances (the two papers cited relate to an industrial application where a main asset—the PikeOS microkernel—of the involved company has been considered). Similar work is by Divakaran et al. [46]. Both efforts have in common that they consider less data-structure intensive properties (e.g. priorities of scheduled tasks in the FreeRTOS realtime operating system) that can be expressed well without several layers of abstraction. Incidentally, they both used the same tool VCC [36], which is well-suited for such lower-level proofs directly on the C code.

However, it is not entirely clear how well it works to develop *abstract* models in retrospect (in the spirit of AFS), as real code tends not to have as clean interfaces (in particular when C is concerned). This is apparent in the Linux kernel as well: For example, UBIFS uses pre-provided internal data structures with cross-pointers and function calls all over the place. The paper [46] actually takes a step towards retrofitting formal models onto existing interfaces, building on previous work [35]. It would be interesting to see this approach to scale to systems of the size and complexity of UBIFS.

[2] https://linuxtesting.org

Chapter 10

Hardware Model

Assumptions about Flash Hardware

Summary. This section formalizes the assumptions about the hardware, captured by the behavior of an abstract interface representing the driver. The API is modeled after the existing Linux Memory Technology Device (MTD) abstraction layer,[1] which essentially exposes the low-level operations of the hardware.

Publications: This chapter is based on [123].

Contents

Although this model is not strictly part of the contribution of this thesis, it is nevertheless important to understand the basis on which the refinement chain is built. It is also the anchor for the verification methodology of power cut safety in Chapter 11.

10.1 State

Flash memory is organized as an array of *physical erase blocks* (PEBs):

state vars (MTD) $\quad pebs : \mathit{Array}\langle \mathit{Peb} \rangle \qquad$ where

data $\mathit{Peb} = \mathtt{mkpeb}(\mathtt{data} : \mathit{Array}_{\mathtt{PEB\_SIZE}}\langle \mathit{Byte} \rangle, \mathtt{fillcount} : \mathit{Nat}, \mathtt{isbad} : \mathit{Bool})$

Each PEB stores a byte-array `data` of fixed length `PEB_SIZE` that is implicitly partitioned into pages of length `EB_PAGE_SIZE`. A PEB stores a page-aligned counter `fillcount` that tracks the part of the block that contains programmed pages, i.e., only data above `fillcount` is known to be empty and can be written to. Note that the fill counter cannot be accessed by software. It is an auxiliary state only used to enforce that pages are written sequentially and never overwritten. PEBs also carry a hardware-supported marker `isbad` that is set explicitly by upper layers after access failures to prevent future usage of the block.

The model maintains the following invariant for all $peb = pebs[i]$ with $\neg\, peb.\mathtt{isbad}$:

invariant (MTD)

$$\begin{aligned} &\mathtt{aligned}(peb.\mathtt{fillcount}) \wedge peb.\mathtt{fillcount} \leq \mathtt{PEB\_SIZE} \\ \wedge\, &\forall\, n.\ peb.\mathtt{fillcount} \leq n < \mathtt{PEB\_SIZE} \rightarrow peb.\mathtt{data}[n] = \mathtt{EMPTY}, \end{aligned} \tag{10.1}$$

[1] http://www.linux-mtd.infradead.org

```
mtd_write(n, offset, len, buf; err)
  precondition   n < #pebs ∧ ¬ pebs[n].isbad ∧ offset + len ≤ PEB_SIZE
                 ∧ pebs[n].fillcount ≤ offset ∧ aligned(offset) ∧ aligned(len)
  choose k with k ≤ len ∧ aligned(n) in
    copy(buf, 0, offset, k; pebs[n].data)
    pebs[n].fillcount := offset + k
    if k = len then err := ESUCCESS else err := EIO

mtd_read(n, offset, len; buf, err)
  precondition n < #pebs ∧ ¬ pebs[n].isbad ∧ offset + len ≤ PEB_SIZE
       { copy(pebs[n].data, offset, 0, len; buf)
         err := ESUCCESS }
    or { err := EIO }

mtd_erase(n; err)
  precondition n < #pebs ∧ ¬ pebs[n].isbad
       { pebs[n] := mkpeb(EMPTY_PEB, 0, false)
         err := ESUCCESS }
    or { err := EIO }

mtd_isbad(n; bad)
  precondition n < #pebs
  bad := pebs[n].isbad

mtd_markbad(n)
  precondition n < #pebs
  pebs[n].isbad := true
```

Figure 10.1: Specification of the Linux MTD driver interface, encoding the expected behavior of the hardware.

where

$$\texttt{aligned}(n) \quad \leftrightarrow \quad n \,\%\, \texttt{EB\_PAGE\_SIZE} = 0$$

ensures that n divides by the size of a physical page. The invariant specifies that the fill count is a multiple of `EB_PAGE_SIZE` and that all bytes above (inclusive) are empty (the constant `EMPTY` equals `0xff` for most real hardware). The invariance of this trivially follows from the preconditions of the operations.

The initial state is given by an arbitrary configuration that satisfies the invariant above: the reason is that one can explicitly model formatting of the device from an unknown state in the erase block management layer, and such an operation has been realized as part of the lower models of Flashix. The format operation simply erases each block of the device and writes some management information to the respective first pages.

initial state (MTD) *pebs* such that (10.1) holds

10.2 Operations

The operations of the model are shown in Figure 10.1. Preconditions ensure proper use of the interface: each block n accessed is valid and not marked bad (for `read`, `write`, and

`erase`). The range *len* to *offset* + *len* of bytes read or written is below the size of the block `PEB_SIZE`. Writing is confined to aligned offsets above `fillcount`.

Hardware failures are modeled by nondeterministically choosing between success or failure by the approach used already in many other layers. An exception is the `write` operation: It chooses a page aligned value k that determines, *how far* the write succeeds. In the best case $k = len$ so that all of the bytes from *buf* are written, whereas in the worst case $k = 0$ none of them reaches the flash medium. Only in the former case, the whole operation can be considered successful, otherwise, an input/output error is returned in *err*.

This model encodes the following assumptions about the hardware:

1. Page writes and block erasure can be viewed as atomic operations. This is represented for write by advancing the counter m and the `fillcount` by `EB_PAGE_SIZE` bytes in each iteration of the loop. For erase, it is apparent by expressing the whole operation as a single statement when `copy` is regarded as atomic.
2. Success of an operation can be recognized, i.e., an error is not returned by mistake. This is modeled by always returning $err := \texttt{ESUCCESS}$ in conjunction with a modification.
3. Conversely, hardware failure can also be detected reliably. In particular, reads that produce garbage can be recognized (leading to $err := \texttt{EIO}$).
4. Bad block handling is reliable, expressed by the absence of returned errors.
5. An unsuccessful page write/block erasure does not modify the state. For write this is achieved by aligning the range copied to flash to page boundaries (`aligned`(k) in the condition of the **`choose`** in Figure 10.1).
6. An unexpected power failure has no *further* effect on the state of the flash device. Figure 10.1 intends to capture the complete behavior as discussed below.

Assumption 5 is not realistic and it is relaxed it to a certain degree by underspecification of upper layers (notably the input/output interface of the erase block management). For example, checksums can be used to recognize certain kinds of data corruption. However, on the level of MTD there is no possibility to express such application-specific concepts. The details have been described in [123].

Assumption 6 is false for flash memory with multi-level cells (MLC), which couples the memory cells of two (or more) flash pages in order to optimize physical space utilization. In this approach, the memory cells are so close together that data in the first page of a pair is not entirely stable when a write to the second one is interrupted by a power cut (see [153, Section 4.1]. However, MLC flash is currently not the target of this project, in fact, support for this kind of memory has just become a beta feature of UBIFS and UBI.

10.3 Power Cuts

From the last assumption of about the behavior of flash hardware, it follows that *power cuts* should have no effect on *pebs*. The recovery operation in this abstract model simply does nothing.

`crash` (MTD) $\quad pebs = pebs'$
`mtd_recovery`( ) {**`skip`**}

10.4 Related Work

Formal models of the Open NAND Flash Interface (ONFI) standard [2] by Butterfield and Catháin [25], Butterfield et al. [26] describe the low level internal structure of flash hardware. A particular relevant aspect here is the addressing of the blocks and their subdivision into so called Logical Units (LUNs) that structure the access to the memory cells and provide opportunities for on-chip parallelism.

In comparison, the Flashix project does not consider the internals of flash hardware as it relies on an abstract specification of the driver interface, which hides all these details. Nevertheless, providing a formal connection between our driver model and the ONFI specifications in [25, 26] could validate our hardware assumptions and demonstrate that it is at least feasible to construct flash memory chips to high assurance guarantees that will ultimately close the remaining gap towards the actual hardware. However, this would clearly be out of scope of this project and is left open for future work.

The formal models [39, 92] discussed before in Section 9.8 do not respect the limitation to sequential writes within an erase block, although non-sequential writes are often not supported by newer ONFI-compliant devices [2].

Chapter 11

Crash-Safe Refinement

Farnsworth: *Behold! A time travelling machine!*
Bender: *Time? I can't go back there!*
Farnsworth: *Ah, but this machine only goes forward in time.*

Summary. This chapter considers systems that may crash during execution. A crash is an event that is triggered asynchronously, aborting the currently executing operation in some intermediate state. Crash safety of such a system is its capability to recover from such an event in a well-specified and practically desirable manner. The approach taken here integrates the verification of crash-recovery correctness into refinement. For the specification of crashes and recovery, first-class modeling constructs are introduced. These are considered by a modified, crashing semantics of machines, given as previously by transition systems. As a consequence, proof principles for trace refinement are directly applicable. Practical verification conditions are given for the general case and for various specializations that reduce the proof effort in practice. Specifically, a technique is developed to propagate the location where a crash must be verified forward in time towards the end of an operation.

Publications: This chapter is based on [55] and the technical report [124].

Contents

This chapter considers data type ASMs that may exhibit *crashes* during execution. A crash is an event that is triggered asynchronously, aborting the currently executing operation in some intermediate state. In the context of file system verification, this typically means unexpected power loss. A crash erases volatile state (i.e., data in main memory such as caches), but the persistent state remains (mostly) unchanged. After a crash, a designated *recovery* operation tries to reconstruct the previous situation.

Intuitively, crash safety states that operations have an observably atomic effect, i.e., an interrupted execution followed by recovery either corresponds to a complete execution of such an operation, or alternatively, partially written data is discarded and it seems as if the operation had never been called in the first place. However, this understanding of crash safety as observable atomicity is insufficient for realistic systems—it is sometimes

not possible or feasible to reconstruct the exact initial or final state during recovery but one that is sufficiently similar to either one. We have already seen in Section 7.6 that this is true already for the topmost POSIX model.

In the flash file system, many intermediate layers expose some effect of a crash that cannot be masked by the recovery because of incomplete information. As a consequence, the effect cannot be masked from the specification either, which motivates our approach to integrate crash safety into refinement: an abstract machine `A` specifies, *to what extent* the corresponding implementation `C` must be able to recover from a crash.

11.1 A Simple Model

A transition system that exhibits crashes is specified by

$$T_\lightning = (St, Init, Cr, Rec, \longrightarrow),$$

with a set St of base states, labels A, and initialization $Init \subseteq St$ as usual. A transitions of the system $\xrightarrow{j,i,o}\!\!\!\!\!\searrow \subseteq St \times (St \uplus St_\lightning)$ may result in a state $St_\lightning := \{s_\lightning \mid s \in St, s \neq \bot\}$ in addition to ordinary ones St. Such a state signifies a partial computation, i.e., one that prompts a subsequent crash. (for this reason, the transition relation is denoted by a "partial" arrow with a half-tip). The effect of the crash is encoded by a relation $Cr \subseteq St_\lightning \times St$ that is immediately followed by recovery $Rec \subseteq St \times St$

In contrast to nontermination $\bot$, a state $s_\lightning$ carries over all the information stored in s, except that it is marked specially. In the interval semantics $I \models p$ of programs, only the last state of a finite interval can be marked in this way, reflecting that after a crash no further computation is possible until the system is rebooted and recovered. If an interval ends in such a state, it cannot possibly be extended by program steps. In this case, condition $I^\uparrow$ is defined to hold too, which means that rule (4.2) from the interval semantics of programs can be used to propagate crashes over sequential program composition $p; q$: If a crash occurs during execution of p then q is not executed at all. In contrast, occurrences of crashed states in machine runs $I \in runs^{T_\lightning}(\underline{j}, \underline{i}, \underline{o})$ are not restricted to final states, because from a crashed state recovery is possible as explained in the following definition, which lifts a transition system $T_\lightning$ to a regular one.

Definition 11.1 (Crashing transition systems)**.** The conventional transition system $T = (St \uplus St_\lightning, A, Init, \longrightarrow)$ induced by $T_\lightning$ lifts intermediate states $s_\lightning$ resulting from partial computations to the level of runs in order to make these observable. The next-state relation $\longrightarrow$ of T lifting $\longrightarrow$ to the complete behavior considers two cases: transitions from regular states to states with a potentially pending crash (first case) and transitions where the crash takes effect and is subsequently recovered (second case).

$$s \xrightarrow{j,i,o} s' \quad \Longleftrightarrow \quad \begin{cases} s \xrightarrow{j,i,o}\!\!\!\!\!\searrow\ s' & \text{where } s \in St \text{ and } s' \in St \uplus St_\lightning \\ (s, s') \in (Cr \fatsemi Rec) & \text{where } s \in St_\lightning \text{ and } s' \in St \end{cases}$$

Note that the "where" clauses are merely there for clarification. They just reflect the domains of the constituent relations and do not impose additional constraints.

By the encoding with regular transition systems, the crashing ones inherit Definition 4.10 of executions and runs, as well as the principles underlying the approach to refinement (see below).

A few things about this definition are noteworthy. The definition does not specify at all the mechanism how such states are generated. It is up to $\longrightarrow$ to determine possible outcomes $s_↯$ as a result of executing this transition partially, whatever that means in practice. When defining $\longrightarrow$ from the interval semantics of a program $I \models p$ later on, it will be natural to take such states from some intermediate $I(k)$, roughly characterized by

$$s \longrightarrow s' \quad \text{when} \quad (s, s') \in [\![p]\!]$$
$$s \longrightarrow s'_↯ \quad \text{when} \quad (s, \cdots, s', \cdots) \models p$$

From the domain of the crash predicate, it is apparent that Cr applies to such marked states only, i.e., as long as the system does not crash, its state is guaranteed to be unaffected. Recovery always follows the crash immediately. It is still possible and useful to consider Rec as part of the normal behavior, i.e., $Rec \subseteq (\longrightarrow)$ makes it necessary to show that the system is crash safe during recovery, too.

The marked states $s_↯$ are the ones just *before* the crash, not after. They are nevertheless visible in the run of T as a possible outcome of $\longrightarrow$: this fact is of course crucial to determine that crashes have occurred in order to align, e.g., the run of a concrete system to an abstract one in refinement, and to propagate submachine crashes to the caller to abort the entire global step. At first, it might seem unintuitive that the uncrashed states are exposed as part of the runs in contrast to states that have been affected by Cr already. The approach taken here has two immediate advantages that are harder to achieve when merging Cr into $\longrightarrow$:

The effect of a crash is separated from its occurrence. Application of Cr is a canonical part of the transition system, independently of how $\longrightarrow$ is realized internally. Conversely, $\longrightarrow$ retains full control over crash atomicity, i.e., where exactly crashes have to be considered. This is useful for modeling and abstraction purposes, leading to two different interpretations of atomicity of operations in Section 11.2.

Crashes are canonically propagated to contexts. When nested machines $\mathtt{M}(\mathtt{X})$ are considered, a distinction must be made whether a crash occurs in the middle of a call $\mathtt{Op}^{\mathtt{X}}$ or outside after an internal step of a program p of $\mathtt{M}$. This aspect is actually independent of applying the effect of the (combined) crash. When these two are kept separate, it is not necessary to modify the nonatomic semantics of programs and submachine calls at all. The notion $I^{\uparrow}$ (of intervals that cannot be extended by program steps) captures nontermination, $\bot$ states, and interrupted states $s_↯$ uniformly, so that the inference rules (4.3) and (4.2) of Definition 4.2 can be reused. The approach furthermore implies that compositionality of refinement as demonstrated in Section 5.3 remains valid for non-recovery steps of the transition system, even if the outcome is a crashed state (first possibility in Definition 11.1).

Refinement between two crashing transition systems is inherited from their encoding into regular ones. Before discussing the mathematical details, we'll illustrate how the runs of $A_↯$ are related to the ones of $C_↯$ in Figure 11.1, which is the analogue to Figure 5.1 in the ordinary case. Several runs are compared, the abstract transitions are depicted at the top, the concrete ones at the bottom: Two regular runs (as_1, as_2, as_3) and (cs_1, cs_2, cs_3) are matched as usual by their inputs and outputs j_k, i_k, o_k. Two crashed runs $(as_1, as_2, as_↯, as', as_4)$ and $(cs_1, cs_2, cs_↯, cs', cs_4)$ (sharing prefixes with the regular ones) are matched so that they have crashed transitions at the same time leading to states $as_↯$ and $cs_↯$, followed by a lockstep recovery. After that, the two systems must be in line again, i.e., the subsequent transitions to as_4 resp. cs_4 have the same labels j_3, i_3, o_3 again.

The formal definition of refinement $A_↯ \sqsubseteq C_↯$ between two crashing transition systems

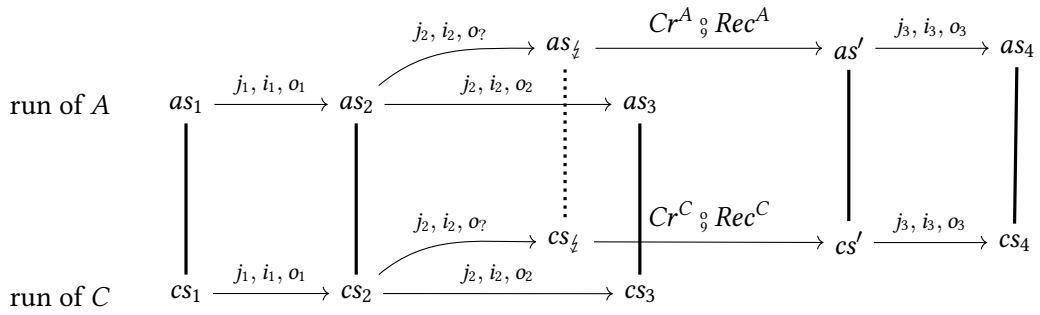


Figure 11.1: Refinement of Runs with Crashes.

needs just one minor modification in comparison to Definition 5.1: the correspondence between two intervals $I^A \sqsubseteq I^C$ must be lifted to deal with crashed states.

Definition 11.2 (Refinement of transition system with crashes)**.** A concrete transition system $C_↯$ that is encoded as C is a correct refinement of an abstract specification $A_↯$ that is encoded as A, iff for all I^C and sequences of observations $\underline{j}, \underline{i}, \underline{o}$

$$I^C \in runs^C(\underline{j}, \underline{i}, \underline{o}) \quad \Longrightarrow \quad \exists\, I^A \in runs^A(\underline{j}, \underline{i}, \underline{o}) \text{ such that } I^A \sqsubseteq I^C$$

where correspondence $I^A \sqsubseteq I^C$ of two intervals requires as before that their lengths is the same and that the concrete system diverges only after the abstract one permits it. The new condition in the third line ($\star$) enforces that the two systems must crash at the same time, because crashed states $St_↯$ are disjoint from regular ones St.

$$\begin{aligned} I^A \sqsubseteq I^C \quad \Longleftrightarrow \quad & \#I^A = \#I^C \text{ and for all } k \leq \#I^A : \\ & I^C(k) = \bot \implies I^A(k) = \bot \text{ and} \\ & I^C(k) \in CSt_↯ \iff I^A(k) \in ASt_↯ \qquad (\star) \end{aligned}$$

Theorem 11.3 (Forward simulation with crashes)**.** *Refinement $A_↯ \sqsubseteq C_↯$ can be proved by an inductive invariant $R \subseteq ASt \times CSt$ with $(\bot, cs) \in R$ for all $cs \in CSt$ that satisfies*

$$\begin{aligned} Init^C &\subseteq Init^A \fatsemi R && && \text{initialization} \\ R \fatsemi (\xrightarrow{j,i,o}_C) &\subseteq (\xrightarrow{j,i,o}_A) \fatsemi R \cup (ASt_↯ \times CSt_↯) && \text{for all } j, i, o && \text{correctness} \\ R \fatsemi (\xrightarrow{j,i,o}_C) \fatsemi Cr^C \fatsemi Rec^C &\subseteq (\xrightarrow{j,i,o}_A) \fatsemi Cr^A \fatsemi Rec^A \fatsemi R && && \text{recovery} \end{aligned}$$

The addition of $ASt_↯ \times CSt_↯$ on the right hand side of the correctness conditions has the effect to drop any requirement for crashed outcomes. These are handled by the recovery condition, which applies when some intermediate state $cs_↯$ before Cr^C is a crash-marked one, forcing its abstract counterpart $as_↯$ to a crash as well (otherwise, these wouldn't be in the domain of the respective Cr relation). That way, one bypasses the need to incorporate crashed states into the proof obligations (similarly to keeping $\bot$ states at the semantic level for Theorem 5.4).

Proof of Theorem 11.3. Analogous to Theorem 5.3 in Section 5.2 for $A \sqsubseteq C$ over the length of the concrete run resp. by diagonaliziation for infinite runs. When the last state is crashed,

the second last transition of the concrete run is examined as well. The recovery condition then gives the desired two new steps for the abstract run. To establish its precondition R, the inductive argument is strengthened so that *all* previously seen pairs of abstract/concrete state are in R when they do not carry the crash marker. □

Remark. In contrast to our initial formulation of the theory in [124] we have elected to mark crashes as part of the states as compared to introducing crashed indices $\mathcal{J}_↯ = \{ j_↯ \mid j \in \mathcal{J} \}$. Both approaches are reasonable to model the crashing semantics. The difference is a rather philosophical one: Crashed indices $j_↯$ seem to suggest that control over whether a crash occurs is dictated to the *caller* just as regular indices, inputs, and outputs. In contrast, the formalization in this thesis makes it clear that it is an intrinsic part of the system itself where crashes may occur (cf. the distinction into white- and black box semantics in the next section). It seems more natural to prevent further execution after a crash by looking at the interval but not the inputs and outputs to keep the notion $I^↑$ and rule (4.2). The approach taken here is also in line with related work by Ntzik et al. [120], which represents crashed states explicitly as described in Section 11.7.

11.2 Atomicity of Crashes

Extending the abstract model of Section 11.1 to machines whose computation is defined by programs with multiple steps and submachine calls raises the question about their atomicity. To what extent can e.g. calls be treated atomically as the semantics in Definition 4.14 suggests? In Figure 11.2, the submachine operation at the bottom does have some intermediate states (grey). Given that `X` is an *abstract* machine, i.e., one that is refined further, it makes sense to interpret `X`-calls as atomic, since proofs for crashed runs of `M(X)` will be simpler. Consequently, only crashes within `X` in the white states should be considered. In contrast, if the submachine corresponds to an implementation, *all* intermediate states (white and grey) are relevant for the analysis, because that is what will eventually run in reality. These different views are captured by two semantics for crash-behavior:

A machine $\mathtt{M}^□$ with "white box" crash behavior permits crashes any time during the execution of its operations. Under certain conditions, such a machine can be reduced to a machine $\mathtt{M}^■$ with "black box" crash behavior: only states in between operations are relevant in order to study crashes. Black box crash behavior of submachines $\mathtt{X}^■$ is what makes it possible to treat calls atomically, in fact, we will treat the operations of any *abstract* model in the refinement hierarchy as atomic, consequently, all specification machines will be considered as black box ones. In the following, let $\mathtt{M}^◪$ denote a machine $\mathtt{M}^□$ or $\mathtt{M}^■$ with either crash behavior.

Example 11.4 (Abstract models are "black box")**.** Take for instance the model of the Linux Memory Technology (MTD) interface shown in Chapter 10 that encodes the expectations to the hardware. It will be regarded as a black box machine, i.e., the lowest layer in the hierarchy is given by $\mathrm{MTD}^■$. The reason why this view is adequate is not just "by convenience"—the model is designed on purpose that the view $\mathrm{MTD}^■$ gives the *complete* observable behavior as discussed in Section 10.3. Notably, the `mtd_write` operation shown in Figure 10.1 already accounts for partial writes that may have been interrupted by a power-cut. Switching to a white box view $\mathrm{MTD}^□$ therefore does not add any additional observations.

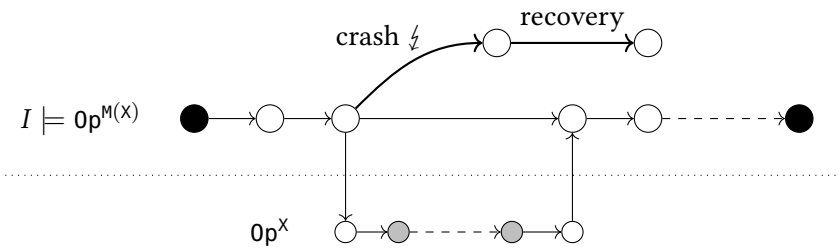


Figure 11.2: Different degrees of atomicity of crash behavior of an operation of machine M(X). The states denoted by circles are generated by the semantics from Section 4.1. $\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ considers the black states only, $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare})$ adds the white states to the behavior and $\mathtt{M}^{\square}(\mathtt{X}^{\square})$ additionally considers the grey states. At the top, an example crash in an intermediate state and subsequent recovery is shown. The corresponding states at the top in the alternate execution are generated by the crashing semantics as defined in Section 11.3.

The same holds for all of the other abstract specification layers in the refinement hierarchy. Moreover, the general approach to model hardware errors by pattern (6.2)

$$\{\ \mathit{body};\ \mathit{err} := \mathtt{ESUCCESS}\ \}\ \ \mathbf{or}\ \ \{\ \mathbf{choose}\ \mathit{err} \in \{\mathtt{EIO}, \cdots\}\ \}$$

induces an "all-or-nothing" effect when the *body* of the operation is considered atomically.

The distinction into black box and white box semantics gives rise to different composition patterns with increasing degree of atomicity (cf. Figure 11.2): Machines $\mathtt{M}^{\square}(\mathtt{X}^{\square})$ consider crashes at any point in time during execution, even in the middle of submachine calls. Such machines correspond to the system that will finally run, i.e., the system whose correctness we are ultimately interested in. Machines $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare})$ view operations of X atomically, however, it is still possible to have a crash in the middle of an M-operation. Machines $\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ consider crashes between M-operations only. From a verification point-of-view such machines are much simpler, since it is possible to reason about their behavior in an entirely atomic setting. Technically, this means that weakest precondition calculus, as presented in Section 3.3, is sufficient for proofs of their correctness.

Example 11.5 (Implementation machines are "white box")**.** Naturally, the power cut analysis of implementation level machines must consider each intermediate state. As already said, this necessitates the white box semantics. For example, during the verification the Virtual File System (VFS) is regarded together with its submachine as the compound system $\mathrm{VFS}^{\square}(\mathrm{AFS}^{\blacksquare})$, whereas the code that will eventually run (i.e., from which code is generated) is the far more complex composed system $\mathrm{VFS}^{\square}(\mathrm{FlashFS}^{\square}(\ldots(\mathrm{MTD}^{\blacksquare})))$—omitting the intermediate layers for brevity.

As one of the consequences, a key feature of the theory should be that refinement remains compositional with respect to submachines. Specifically, it must be possible to replace $\mathrm{AFS}^{\blacksquare}$ in its context $\mathrm{VFS}^{\square}(\_)$ by the concrete $\mathrm{FlashFS}^{\square}(\ldots(\mathrm{MTD}^{\blacksquare}))$ without compromising the correctness of the whole system with respect to the top-level POSIX specification even in the presence of crashes.

The point made by the example is formally captured by the following theorem (proved in Section 11.3 in this chapter on page 133).

Theorem (Compositionality under crashes)**.** *A submachine with crashes can be substituted in a context,* $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\blacksquare}$ *implies* $\mathtt{M}^{\square}(\mathtt{A}^{\blacksquare}) \sqsubseteq \mathtt{M}^{\square}(\mathtt{C}^{\blacksquare})$ *(and similarly for the other combinations of white/black box machines).*

Generally, while the white box semantics $\mathtt{M}^{\square}$ is adequate, it is not very convenient to directly verify such machines. The proofs need to refer to the intermediate states of the operations arising from the fine-grained semantics (cf. Figure 11.2). We therefore seek ways to switch from a white box to a black box view whenever possible, because a formal proof for the latter has to consider *significantly* less states. Under certain conditions called "crash neutrality" this switch is indeed possible, as expressed by the following theorem (proved in Section 11.6 on page 141).

Theorem (Reduction)**.** $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare}) \equiv \mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ *if* $\mathtt{X}^{\blacksquare}$ *is crash neutral and* $\mathtt{M}^{\square}$ *has RAM state only.*

The intuition behind this reduction is subsumption of crash behavior: Crashes in an intermediate state s_1 taken from the system's execution are mapped to a different state s_2 that can be recovered alike. Since there may be more possibilities to recover s_2 in comparison to s_1 but not less, it is sufficient to consider a crash in the state s_2 only. Crash neutrality is a criterion that guarantees that *accessible* candidate states s_2 exist that can always be taken from the black box semantics: Intuitively, the criterion says that a partial execution of an operation can always be extended towards a complete one without altering the outcome of crash-recovery. The construction relies on nondeterministic errors as exhibited by the submachine $\mathtt{X}^{\blacksquare}$ according to (6.2) to ensure that such completions are always possible, and on the fact that in the state of all implementation components $\mathtt{M}^{\square}$ is stored in main memory and will therefore be arbitrary after a crash anyway.

11.3 Crash-Aware Machines

This section defines crash-aware machines and submachine composition in relation to the concepts of the previous sections. It is stated how refinement can in principle be established. Proof obligations for the general case are postponed to Section 11.5 and for some special cases to Section 11.6.

Definition 11.6 (Data type ASM with crash behavior)**.** A data type ASM with crash behavior $\mathtt{M}^{\blacktriangleleft} = (\underline{x}\colon \mathit{St}, \mathtt{Init}, \{\mathtt{Op}\}_{j\in\mathcal{J}}, \mathtt{Cr})$ has an additional predicate $\mathtt{Cr}\colon \mathit{St} \times \mathit{St} \to \mathit{Bool}$ describing possible state transitions triggered by a power cut. Immediately afterwards (and only then) a designated operation $\mathtt{Rec} = \mathtt{Op}_{\mathrm{rec}}$ with index $\mathrm{rec} \in \mathcal{J}$ is called implicitly in order to restore a consistent state.

We define the semantics of machines $\mathtt{M}^{\blacktriangleleft}$ in terms of a modified atomic semantics $[\![\mathtt{Op}]\!]^{\blacktriangleleft}$ of operations, which in turn refers to a potentially crashing atomic semantics $[\![p]\!]^{\blacktriangleleft}$ of programs shown in Definition 11.7 below.

The *white box* semantics $[\![p]\!]^{\square}$ exposes crashes in each intermediate state of the execution of a program. It incorporates in addition to complete behaviors those successor states $s'_{\lightning}$ that can be generated from a partial execution $(s, \cdots, s') \fatsemi I$ where the remaining interval I is discarded.

The *black box* semantics $[\![p]\!]^{\blacksquare}$ takes its states just from the regular atomic one. Specifically, the second clause includes immediate crashes $(s, s_{\lightning})$ that happen before the program is even started, whereas the third clause includes final states of p.

Definition 11.7 (Crashing program semantics)**.**

$$[\![p]\!]^{\square} = [\![p]\!] \cup \{(s, s'_{\lightning}) \mid \exists I.\ (s, \cdots, s') \fatsemi I \models p\}$$
$$[\![p]\!]^{\blacksquare} = [\![p]\!] \cup \{(s, s_{\lightning}) \mid s \in St\} \cup \{(s, s'_{\lightning}) \mid (s, s') \in [\![p]\!]\}$$

such that $s, s' \neq \bot$ is required for each occurrence in the definition.

Remark. Since the black box semantics is an artifact introduced to express certain facts within the theory, one could argue that including both possibilities of an immediate crash and one after fully executing p is redundant: either one might suffice because when sequences of operations $\mathtt{Op}_{j_1}; \mathtt{Op}_{j_2}$ are considered as part of runs, it is irrelevant whether the crash in the intermediate state is generated from the final state of $\mathtt{Op}_{j_1}$ or the initial state of $\mathtt{Op}_{j_2}$. We will see that immediate crashes $(s, s_{\lightning})$ lead to a simpler argument later on (Lemma 11.21), whereas including the final states is in fact necessary when submachines executions are considered in Lemma 11.11.

The lifting to operations $\mathtt{Op} = (\mathtt{pre}, in, p, out)$ is analogous to Definition 4.8, except that the domain $[\![\mathtt{Op}^{\blacksquare}]\!] \subseteq A_{in} \times St \times (St \uplus St_{\lightning}) \times A_{out}$ of operations now reflects the possibly crashed outcomes although starting states are still confined to St to capture that regular operations are prohibited in crashed states. We repeat the complete definition:

$$(i, s, s', o) \in [\![\mathtt{Op}]\!]^{\blacksquare}$$
$$\iff \begin{cases} (s(in \mapsto i), s') \in [\![p]\!]^{\blacksquare} & s(in \mapsto i) \models \mathtt{pre} \\ s' = \bot & \text{otherwise} \end{cases}$$

such that $o = s'(out)$ if s' is a regular uncrashed state different from $\bot$, otherwise o is arbitrary.

Definition 11.8 (Semantics of machines with crashes)**.** The semantics of a machine exhibiting crashes $\mathtt{M}^{\blacksquare} = (\underline{x}\colon \mathtt{St}, \mathtt{Init}, \mathtt{Cr}, \{\mathtt{Op}\}_{j \in \mathcal{J}})$ is given by the crashing transition system $M_{\lightning} = (St, Init, Cr, Rec, \longrightarrow)$ such that

$$s \xrightarrow{j,i,o} s' \quad \iff \quad (i, s, s', o) \in [\![\mathtt{Op}_j]\!]^{\blacksquare},$$

where initialization, crash and recovery are defined by

$$\begin{aligned} Init &= \{s \in St \mid s \models \mathtt{Init}\} && \text{(as in Definition 4.11)} \\ Cr &= \{(s_{\lightning}, s') \mid (s, s') \in [\![\mathtt{Cr}]\!]\} && \text{(adding the crash marker)} \\ Rec &= [\![\mathtt{Rec}]\!] \end{aligned}$$

and $\mathtt{Rec} = \mathtt{Op}_{\mathrm{rec}}$ for the special index $\mathrm{rec} \in \mathcal{J}$.

The regular transition system M that assigns these machines a semantics is given by embedding Definition 11.1, i.e., by $M := (St \uplus St_{\lightning}, Init, \longrightarrow \cup (Cr \fatsemi Rec))$.

In Section 11.1 it is assumed that $Rec \subseteq St \times St$ is a binary relation on states and here Rec has no inputs and no outputs so that the typing works out in the above definition, although it makes sense to consider both for a submachine X that is recovered as part of an outer context. Such an extension is not hard to define but this is not considered here to keep the presentation simpler.

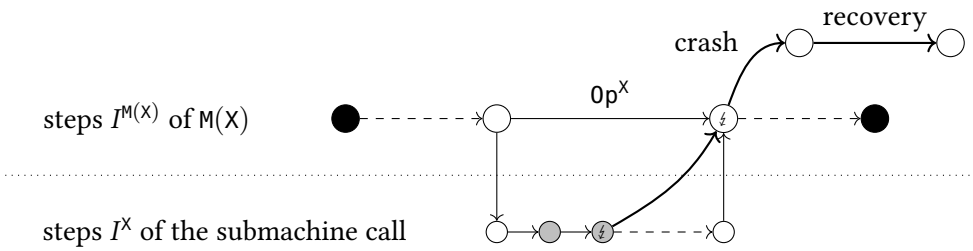


Figure 11.3: Propagation of a submachine crash to the caller. The fat arrow denotes the states encountered. At the bottom, a crash occurs in the intermediate submachine state marked by $\lightning$, which cancels the regular execution and immediately jumps to the end of the operation in the caller into a state that also recieves the crash marker. Subsequently, the crash and recovery cycle is triggered as part of the combined $\mathtt{M}(\mathtt{X})$ run.

11.4 Submachines and Modularity

We now discuss submachine composition of crashing machines and discusses how the compositionality result (Theorem 5.9) can be retained in a very straight-forward way.

Definition 11.9 (Submachine composition in the presence of crashes)**.** Submachine composition $\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ of two machines with crash behavior is defined by

$$\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare}) := (\underline{mx}, \underline{xx} \colon \mathit{St}^{\mathtt{M}}, \mathit{St}^{\mathtt{X}}, \quad \mathtt{Init}^{\mathtt{M}} \wedge \mathtt{Init}^{\mathtt{X}}, \quad \mathtt{Cr}^{\mathtt{M}} \wedge \mathtt{Cr}^{\mathtt{X}}, \quad \{\mathtt{Op}_k^{\mathtt{M}}\}_{k \in K})$$

as an extension of Definition 4.13, additionally combining the crash predicates and defining compound recovery as

$$\mathtt{Rec}^{\mathtt{M}(\mathtt{X})}() \ \{\ \mathtt{Rec}^{\mathtt{X}}();\ \mathtt{Rec}^{\mathtt{M}}()\ \},$$

where $\mathtt{Rec} \equiv \mathtt{Op}_{\mathrm{rec}}$ so that the submachine is recovered *first*. This means that when $\mathtt{Rec}^{\mathtt{M}}$ is started, $\mathtt{X}^{\blacksquare}$ is up and working, in particular, the invariants of $\mathtt{X}^{\blacksquare}$ are reestablished and the recovery of $\mathtt{M}^{\blacksquare}$ can call ordinary operation of $\mathtt{X}^{\blacksquare}$.

The states $(xs_{\lightning} \oplus ms) = (xs \oplus ms_{\lightning}) = (xs \oplus ms)_{\lightning}$ are identified so that it is irrelevant, which machine has prompted a crash. This lifts a crash that has ooccured in a (white box) submachine call to the caller as shown in Figure 11.3: In the intermediate grey state marked by $\lightning$ of the submachine execution subsequent computation is prevented, hence this state is taken as the outcome of $\mathtt{Op}^{\mathtt{X}}$. The crash marker is propagated to the combined white state in the interval $I^{\mathtt{M}(\mathtt{X})}$ of the caller. With the same mechanism, the surrounding $\mathtt{M}$ operation is also aborted. The subsequent transition picks up the resulting state, applies the crash predicate and calls the recovery operation (the transition is broken down into two steps here).

Theorem 11.10 (Compositionality in the presence of crashes)**.**
Let $\mathtt{C} \in \{\mathtt{C}^{\square}, \mathtt{C}^{\blacksquare}\}$ *and* $\mathtt{A} \in \{\mathtt{A}^{\square}, \mathtt{A}^{\blacksquare}\}$. *Then* $\mathtt{A} \sqsubseteq \mathtt{C}$ *implies* $\mathtt{M}^{\square}(\mathtt{A}) \sqsubseteq \mathtt{M}^{\square}(\mathtt{C})$ *and* $\mathtt{M}^{\blacksquare}(\mathtt{A}) \sqsubseteq \mathtt{M}^{\blacksquare}(\mathtt{C})$.

Proof. The proof reconsiders the extraction of a submachine execution from the steps of a program in Lemma 4.22 and the high-level steps from Section 5.3 in turn. We show which arguments uphold and where additional cases arise due to power cuts. The critical parts are related to the extraction of a submachine execution from the context.

Lemma 11.11 (Submachine execution with crashes)**.** *For a composed system* $\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ *a partial white box execution* $I_1^{\mathtt{X}} \oplus I_1^{\mathtt{M}} \fatsemi I_2^{\mathtt{X}} \oplus I_2^{\mathtt{M}}, \underline{j}_1\underline{j}_2, \underline{i}_1\underline{i}_2, \underline{o}_1\underline{o}_2 \models p$ *which discards the second part implies the existence of* $I_1^{\mathtt{X}\prime}$ *such that* $I_1^{\mathtt{X}\prime} \in execs_\tau^{\mathtt{X}^{\blacksquare}}(\underline{j}_1, \underline{i}_1, \underline{o}_1)$ *where* $I_1^{\mathtt{X}\prime}(k) = I_1^{\mathtt{X}}(k)_{\lightning}$ *for all states* k *including and after the last submachine call and the states before this call are equal in both intervals.*

The lemma ensures that for an intermediate crash in the supermachine there is a way to adapt the execution of the submachine so that $I_1^{\mathtt{X}\prime}.last$ carries a crash marker.

Proof. By Lemma 4.22, $I_1^{\mathtt{X}} \fatsemi I_2^{\mathtt{X}}$ is a submachine execution, which can be split into its constituents according to Lemma 4.23, specifically, $I_1^{\mathtt{X}} \in execs^{\mathtt{X}^{\blacksquare}}(\underline{j}_1, \underline{i}_1, \underline{o}_1)$. We replace the last submachine call in $\underline{j}_1$ by a crashed one. When $\mathtt{X}^{\square}$ is a white box machine, we therefore take an empty remainder I in Definition 11.7 for the call $\mathtt{Op}_{j_k}^{\mathtt{X}}$. When $\mathtt{X}^{\blacksquare}$ is a black box machine, it can also crash after completion of the operation as discussed alongside Definition 11.7. All steps after k are stutter transitions, which preserve the crashed state. □

Purpose of Step 1 is to fix execution fragments $I_k^{\mathtt{C}} \in execs_\tau^{\mathtt{C}^{\blacksquare}}(\underline{j}_k, \underline{i}_k, \underline{o}_k)$ for the submachine $\mathtt{C}^{\blacksquare}$ of the concrete system $\mathtt{M}^{\blacksquare}(\mathtt{C}^{\blacksquare})$. The interval $I_k^{\mathtt{C}}$ records the submachine calls in the k-th step of the global interval executing some program p_j belonging to the called $\mathtt{M}^{\blacksquare}$ operation in that step.

We adapt characterization (5.3) in the high-level proof Step 1 in Section 5.3 with a case for crash and recovery:

$$s \xrightarrow{j,i,o} s' \quad \Longleftrightarrow$$

$$\exists\, I^{\mathtt{C}}, I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o}.\ \begin{cases} I^{\mathtt{C}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p_j, & \text{if } s \neq \bot, s(in \mapsto i) \models \mathtt{pre}_j \\ I^{\mathtt{C}} \oplus I^{\mathtt{M}} \fatsemi \cdots,\ \underline{j}.., \underline{i}.., \underline{o}.. \models p_j, & \text{for a supermachine crash} \\ I^{\mathtt{C}} \oplus I^{\mathtt{M}}, \underline{j}, \underline{i}, \underline{o} \models p_{\mathrm{rec}}, & \text{if } s \text{ carries a crash marker} \\ I^{\mathtt{C}} = I^{\mathtt{M}} = (\bot), & \text{otherwise} \end{cases} \qquad (\star)$$

where $(I^{\mathtt{C}} \oplus I^{\mathtt{M}}).first = s(in \mapsto i)$, $s' := (I^{\mathtt{C}} \oplus I^{\mathtt{M}}).last$, and $o := s'(out)$ when $s' \neq \bot$ in the regular case as before. The first line captures submachine crashes, too, when the last state of the combined interval s' is already crash-marked.

The two additional cases ($\star$) in the second and third line deal with crashes. The second line mirrors the white box semantics of the surrounding surrounding operation: from Definition 11.7, there is a dropped remainder .. so that only the prefix $I^{\mathtt{C}} \oplus I^{\mathtt{M}}$ is executed. In this case, the successor state $s' := (I^{\mathtt{C}} \oplus I^{\mathtt{M}}).last_{\lightning}$ of the transition is lifted to a crash marker. The third line executes a recovery operation, in which case the initial state of the intervals are affected by a power cut, i.e., we have $(s, (I^{\mathtt{C}} \oplus I^{\mathtt{M}}).first) \in Cr^{\mathtt{M}(\mathtt{C})}$. In the second case, Lemma 11.11 ensures that the extracted interval $I^{\mathtt{C}}$ can be mapped to an execution $I^{\mathtt{C}\prime}$ of $\mathtt{C}^{\blacksquare}$ that also has a crashed marker at the end.

Step 1 subsequently concatenates the $I_k^{\mathtt{C}}$ to an $\mathtt{C}^{\blacksquare}$ run $I_*^{\mathtt{C}} = I_1^{\mathtt{C}} \fatsemi I_2^{\mathtt{C}} \fatsemi \cdots$. We must show that this composition is well-defined, i.e., the respective last state $I_k^{\mathtt{C}}.last$ coincides with the first state $I_{k+1}^{\mathtt{C}}.first$ of the subsequent interval. The only critical case is to pair crash-marked states, which is provisioned by taking the modified $I_k^{\mathtt{C}\prime}$ from Lemma 11.11. □

Step 2 remains unaffected. In Step 3, substitution of submachine calls by Lemma 5.6 must treat crashed states analogously to divergence. Finally, in Step 4 we must pay attention to remove the extra crash markers introduced by Lemma 11.11 to refit the abstract interval onto the program.

11.5 General Proof Methods

Refinement $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\blacksquare}$ for two machines follows from Theorem 11.3 for transition systems by a forward simulation $\mathtt{R}$. The initialization, applicability, and correctness conditions are the same as in Theorem 5.4. However, for white box machines, the recovery condition cannot be expressed directly in the weakest-precondition calculus, which cannot reason about intermediate states that are generated by the white box semantics of programs $[\![p]\!]^{\square}$. Simply substituting the definitions gives for the general case the following unwieldy criterion as a starting point to develop a proof strategy:

$$[\![\mathtt{R}]\!] \fatsemi [\![\mathtt{Op}_j^{\mathtt{C}}]\!]^{\blacksquare} \fatsemi \underline{[\![\mathtt{Cr}^{\mathtt{C}}]\!] \fatsemi [\![\mathtt{Rec}^{\mathtt{C}}]\!]} \subseteq [\![\mathtt{Op}_j^{\mathtt{A}}]\!]^{\blacksquare} \fatsemi \underline{[\![\mathtt{Cr}^{\mathtt{A}}]\!] \fatsemi [\![\mathtt{Rec}^{\mathtt{A}}]\!]} \fatsemi [\![\mathtt{R}]\!]. \tag{11.1}$$

Note that the inputs and outputs of the operations are suppressed here to simplify the presentation.[1]

There are a number of special cases of the different combinations of refinement between black and white box machines that are worth considering. Immediate results are (without proof):

Proposition 11.12. $\mathtt{A}^{\square} \sqsubseteq \mathtt{A}^{\blacksquare}$ *always holds, since every crashed outcome of the black box semantics is contained in the white box one (cf. Definition 11.7).*

This means we can take the more fine grained white box machine as a specification of the corresponding coarse grained black box machine.

Proposition 11.13. *Instead of proving* $\mathtt{A}^{\square} \sqsubseteq \mathtt{C}^{\square}$ *it is sufficient to prove* $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\square}$*, by transitivity of refinement and the previous proposition.*

This confirms the intuition that one can regard abstract machines as atomic if desired.

The next three sections discuss three specific combinations:

For the most general and complex case, a white box refinement $\mathtt{A}^{\square} \sqsubseteq \mathtt{C}^{\square}$, formula (11.1) can be expressed as a safety property using temporal program logics as described in Section 11.5.2.

Purely black box refinements $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\blacksquare}$ can of course be proved using weakest precondition calculus. In that case, a separate condition about recovery that is independent of the preceding operation is sufficient as shown below in Lemma 11.16. Such refinements are subject to Section 11.5.3.

Ultimately, we are interested in proving refinements of the form $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\square}(\mathtt{X}^{\blacksquare})$ as outlined in the Examples 11.4 and 11.5 in Section 11.2: An abstract specification $\mathtt{A}$ is refined to the system composed of an implementation part $\mathtt{C}$ and a submachine specification $\mathtt{X}$. Section 11.6 demonstrates a verification strategy for this case, which is used in the Flashix project, building on the results for black box refinements.

11.5.1 Breaking down Crash Refinement

The first step to break (11.1) down is to characterize pairs of abstract and concrete states that are recovered correctly as required by the refinement condition, i.e., the ones that are passed through into underlined part in (11.1) so that $\mathtt{R}$ holds for the outcome. Correct recovery for a refinement $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\blacksquare}$ is formally specified by the semantic judgement

$$as \sqsubseteq_{P\lightning} cs \quad \Longleftrightarrow \quad \{(as_{\lightning}, cs_{\lightning})\} \fatsemi Cr^{C} \fatsemi Rec^{C} \subseteq Cr^{A} \fatsemi Rec^{A} \fatsemi P \tag{11.2}$$

[1] Formally, inputs and outputs are lifted over relational composition by defining, e.g., $[\![\mathtt{R}]\!] \fatsemi [\![\mathtt{Op}_j]\!] = \{(i, as, cs', o) \mid \exists\, cs.\ (as, cs) \in [\![\mathtt{R}]\!] \text{ and } (i, cs, cs', o) \in [\![\mathtt{Op}_j]\!]\}$.

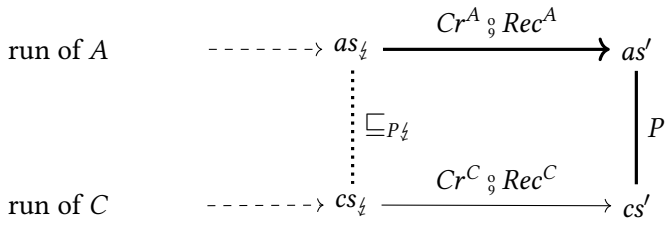


Figure 11.4: Correct recovery for $as_↯$ and $cs_↯$ establishes the diagram shown by constructing the fat arrow at the top.

over states as and cs of the two machines. It is parametrized (implicitly) by the two machines and by a condition $P \subseteq ASt \times CSt$ that generalizes the postcondition $[\![R]\!]$ of (11.1) and guarantees that as and cs can be crash-recovered in lockstep to some states as' and cs' that satisfy $(as', cs') \in P$. The commuting diagram for correct recovery is illustrated by Figure 11.4. It establishes the fat arrow at the top as well as the relation P on the right.

Proposition 11.14 (Correct recovery). *For a refinement between two machines* $\mathtt{A}^{◪} \sqsubseteq \mathtt{C}^{◪}$ *correct recovery from two states* $\underline{ax}$, $\underline{cx}$ *and a desired postcondition* $\mathtt{P} : \mathtt{St}^{\mathsf{A}} \times \mathtt{St}^{\mathsf{C}} \to \mathtt{Bool}$ *can be proved as follows:*

$$\begin{aligned}&\mathtt{correct\text{-}recovery}(\underline{ax}_1, \underline{cx}_1, \mathtt{P}) :=\\ &\quad \forall\, \underline{cx}_2.\ \mathtt{Cr}^{\mathsf{C}}(\underline{cx}_1, \underline{cx}_2)\\ &\qquad \to \langle\!|\mathtt{Rec}^{\mathsf{C}}(; \underline{cx}_2)|\!\rangle \left(\exists\, \underline{ax}_2.\ \mathtt{Cr}^{\mathsf{A}}(\underline{ax}_1, \underline{ax}_2) \wedge \langle \mathtt{Rec}^{\mathsf{A}}(; \underline{ax}_2)\rangle\ \mathtt{P}(\underline{ax}_2, \underline{cx}_2)\right).\end{aligned}$$

This formula introduces the states $\underline{ax}_2$, $\underline{cx}_2$ that are the outcomes of applying the crash predicates such that for each crash transition of the concrete system there must be a corresponding abstract one. The final requirement is that from these two states, the systems can recover to a state where the condition $\mathtt{P}$ holds, encoded by weakest precondition modalities similarly to the correctness condition for regular refinement in Theorem 5.4.

We use the predicate `correct-recovery` from Proposition 11.14 as a building block for the syntactic recovery conditions in the later sections. The remaining difficulty will therefore be how to determine the intermediate states $\underline{ax}_1$ and $\underline{cx}_1$ from the black or white box semantics of the operations in which the crash and subsequent recovery is started.

11.5.2 Temporal Logic Proofs

The highly expressive logic RGITL [145] features a calculus that can reason about the intermediate states of the fine-grained semantics $I \models p$ of programs. Besides temporal logic operations such as $\Box$ ("always") and $\Diamond$ ("eventually"), the logic supports existential path quantifier, written $\mathbf{E}\ \underline{x}.\ \varphi$, that binds variables $\underline{x}$ to new values throughout an interval:

$$I \models \mathbf{E}\ \underline{x}.\ \varphi \quad \iff \quad \text{there is a sequence of values } \underline{\boldsymbol{a}} \text{ such that } I(\underline{x} \mapsto \underline{\boldsymbol{a}}) \models \varphi.$$

We'll use existential path quantification to characterize partial executions of the white box semantics and we can encode the refinement proof obligation (11.1) as follows.

Proposition 11.15 (Recovery). *The recovery condition of any crash safe refinement* $\mathtt{A}^{◪} \sqsubseteq \mathtt{C}^{◪}$

is encoded by the syntactic proof obligation:

$$\begin{aligned} &\mathsf{R}(\underline{ax}_0, \underline{cx}_0), \quad (\mathbf{E}\ \underline{cx}.\ \underline{cx} = \underline{cx}_0 \wedge [\mathsf{Op}^{\mathsf{C}}]_{\underline{cx}} \wedge \Diamond\ \underline{cx} = \underline{cx}_1) \\ &\vdash\ \exists\ \underline{ax}_1. \quad (\mathbf{E}\ \underline{ax}.\ \underline{ax} = \underline{ax}_0 \wedge [\mathsf{Op}^{\mathsf{A}}]_{\underline{ax}} \wedge \Diamond\ \underline{ax} = \underline{ax}_1) \\ &\qquad\qquad \wedge\ \mathtt{correct\text{-}recovery}(\underline{ax}_1, \underline{cx}_1, \mathsf{R}) \end{aligned} \tag{11.3}$$

Starting initially with the states $\underline{ax}_0$ and $\underline{cx}_0$, two executions are determined. The antecedent fixes a trace where $[\mathsf{Op}^{\mathsf{C}}]_{\underline{cx}}$ executes on the variables $\underline{cx}$ that are initially equal to $\underline{cx}_0$ and eventually, in an intermediate state, they are equal to $\underline{cx}_1$. In the consequent, we have to find a state $\underline{ax}_1$ taken sometime from the abstract execution $[\mathsf{Op}^{\mathsf{A}}]_{\underline{ax}}$. The final, condition $\mathtt{correct\text{-}recovery}(\underline{ax}_1, \underline{cx}_1, \mathsf{R})$ ensures that these states can be recovered back to R (see Proposition 11.14).

Proof obligation (11.3) is hardly practical. Related work discussed in Section 11.7 addresses the general case by specialized calculi. The next two sections describe two complementary techniques to prove crash-safe refinement in a simple way.

11.5.3 Black Box Refinements

Purely black box refinements $\mathsf{A}^{\blacksquare} \sqsubseteq \mathsf{C}^{\blacksquare}$ just refer to the atomic semantics of programs and thus it is possible to derive proof obligations based on weakest-precondition modalities. It is then straight-forward to separate the crash and subsequent recovery from the preceding operation, so that a crash and its recovery can be proved in isolation. Hence, it suffices to prove $\mathtt{correct\text{-}recovery}(\underline{ax}, \underline{cx}, \mathsf{R})$ (cf. Proposition 11.14) for all states with $\mathsf{R}(\underline{ax}, \underline{cx})$ in addition to the regular forward simulation conditions. Spelled out in full, we have:

Lemma 11.16. *For a black box refinement $\mathsf{A}^{\blacksquare} \sqsubseteq \mathsf{C}^{\blacksquare}$ the following simple, syntactic recovery proof obligation is sufficient, in addition to the ones for standard forward simulation as given in Theorem 5.4 in Section 5.2.*

$$\begin{aligned} &\mathsf{R}(\underline{ax}_1, \underline{cx}_1) \wedge \mathsf{Cr}^{\mathsf{C}}(\underline{cx}_1, \underline{cx}_2) \qquad \text{isolated recovery} \\ &\vdash \langle\!|\mathsf{Rec}^{\mathsf{C}}(; \underline{cx}_2)|\!\rangle\ \exists\ \underline{ax}_2.\ \mathsf{Cr}^{\mathsf{A}}(\underline{ax}_1, \underline{ax}_2) \wedge \langle \mathsf{Rec}^{\mathsf{A}}(; \underline{ax}_2)\rangle\ \mathsf{R}(\underline{ax}_2, \underline{cx}_2) \end{aligned} \tag{11.4}$$

Proof Sketch. The reason why we can afford to split off crash and recovery completely from the correctness condition of forward simulation (Theorem 5.4) of the preceding operations is that we have the strong guarantee that R holds, even after a crashed execution of $[\![\mathsf{Op}_j^{\mathsf{A}}]\!]^{\blacksquare}$ resp. $[\![\mathsf{Op}_j^{\mathsf{C}}]\!]^{\blacksquare}$. □

The sequent (11.4) is similar to the correctness condition for regular operations, as for every outcome of the concrete recovery after a given crash, there must be a state $\underline{ax}$ to which the abstract system can mirror the crash and subsequently recovery (again, the variables in $\mathsf{R}(\underline{ax}, \underline{cx})$ in the postcondition refer to the final state after the transition).

11.6 Crash Neutrality and Reductions

The approach presented in this section permits to verify refinements $\mathsf{A}^{\blacksquare} \sqsubseteq \mathsf{C}^{\square}(\mathsf{X}^{\blacksquare})$ by transitivity using the intermediate black box system $\mathsf{C}^{\blacksquare}(\mathsf{X}^{\blacksquare})$. The results of this section are sufficient criteria that imply $\mathsf{C}^{\blacksquare}(\mathsf{X}^{\blacksquare}) \sqsubseteq \mathsf{C}^{\square}(\mathsf{X}^{\blacksquare})$, which is established in two steps.

- *Crash-neutrality* ensures that the black and white box semantics coincide, i.e., for a crash neutral machine $\mathsf{X}^{\square} \equiv \mathsf{X}^{\blacksquare}$ holds. In the Flashix file system, all abstract intermediate layers satisfy this criterion.

- Crash neutrality for composed system $\mathtt{C}^{\square}(\mathtt{X}^{\blacksquare})$ then follows, if $\mathtt{C}$ is an implementation level machine (to be made precise).

As a consequence, this scheme propagates one step upwards the refinement hierarchy. It can be lifted from the hardware level MTD (which is crash neutral) to every intermediate refinement layer by proving the simple condition for crash neutrality over just specification level machines. As a consequence, we always have black box refinements and the weakest-precondition based verification method of Section 11.5.3 applies.

The idea behind crash neutrality is based on subsumption of crash/recovery behavior, written as $s_1 \sqsubseteq_{\lightning} s_2$, which ensures that a crash and subsequent recovery in the state s_1 can be simulated from s_2 too with exactly the same outcome. State s_1 is therefore irrelevant for the analysis:

$$s_1 \sqsubseteq_{\lightning} s_2 \quad \Longleftrightarrow \quad \text{for all } s' : \ (s_1, s') \in Cr \fatsemi Rec \implies (s_2, s') \in Cr \fatsemi Rec$$

Notation $s_1 \sqsubseteq_{\lightning} s_2$ intentionally resembles the one of correct recovery (11.2), as it is an instance of $as_1 \sqsubseteq_{P\lightning} cs_2$ for taking the same machine for both the abstract and concrete, and setting the correspondence P for the outcome to equality. Note that $\sqsubseteq_{\lightning}$ is reflexive. We say that s_1 is crash equivalent to s_2 when the subsumption criterion holds in both directions.

Crash neutrality ensures that each intermediate state taken from the execution of a program of a white box machine is crash equivalent to some final state already contained in the black box semantics.

Definition 11.17 (Crash neutrality)**.** A machine $\mathtt{M}^{\blacktriangle}$ is *crash neutral*, if partially executed operations produce states only whose crash behavior is subsumed by some final states of the atomic semantics. For each operation $\mathtt{Op}_j$, input i, dropped output o_1, initial state s, and intermediate state s_1:

$$(i, s, s_1, o_1) \in [\![\mathtt{Op}_j]\!]^{\blacktriangle} \implies \text{there is } s_2, o_2.\ (i, s, s_2, o_2) \in [\![\mathtt{Op}_j]\!] \text{ and } s_1 \sqsubseteq_{\lightning} s_2,$$

or suggestively expressed by

$$[\![\mathtt{Op}_j]\!]^{\blacktriangle} \fatsemi [\![\mathtt{Cr}]\!] \fatsemi [\![\mathtt{Rec}]\!] \subseteq [\![\mathtt{Op}_j]\!] \fatsemi [\![\mathtt{Cr}]\!] \fatsemi [\![\mathtt{Rec}]\!],$$

again ignoring some mismatches between the domains of the relations as in (11.1).

Note that the right hand side of the set inclusion refers to the regular atomic semantics of operations $[\![\mathtt{Op}_j]\!]$—we require a *complete* execution of the operation, not just an atomic crash $[\![\mathtt{Op}_j]\!]^{\blacksquare}$ that might happen right at the start the operation, too. This means that Definition 11.17 is a stronger criterion than (11.1):

Lemma 11.18 (Reduction)**.** *If* $\mathtt{M}^{\square}$ *is crash neutral then* $\mathtt{M}^{\blacksquare} \sqsubseteq \mathtt{M}^{\square}$ *and hence* $\mathtt{M}^{\blacksquare} \equiv \mathtt{M}^{\square}$ *by Proposition* 11.12*.*

Proof. From Definition 11.17 and (11.1) when $\mathtt{R}$ is taken as the identity relation. □

So far, we have no easy way to demonstrate that a machine is crash neutral. We now derive suitable syntactic proof obligations for black box machines $\mathtt{M}^{\blacksquare}$. There are two possibilities for the constituent $(s, s'_{\lightning}) \in [\![\mathtt{Op}_j]\!]^{\blacksquare}$ according to Definition 11.7. The case where $[\![\mathtt{Op}_j]\!]^{\blacksquare}$

is executed fully is trivial, because that maps directly to the right hand side of the set inclusion. Therefore, it remains to show that for each initial state s, there is some state s' such that $(s, s') \in [\![\mathtt{Op}_j]\!]$ that subsumes s. Hence, for $\mathtt{M}^{\blacksquare}$, Definition 11.17 is equivalent to

$$[\![\mathtt{Cr}]\!] \fatsemi [\![\mathtt{Rec}]\!] \subseteq [\![\mathtt{Op}_j]\!] \fatsemi [\![\mathtt{Cr}]\!] \fatsemi [\![\mathtt{Rec}]\!]. \tag{11.5}$$

(again suppressing the in- and outputs of $\mathtt{Op}_j$). The semantic criterion of crash neutrality can be proved as follows.

Lemma 11.19 (Crash neutrality for blackbox ASMs.)**.** $\mathtt{M}^{\blacksquare}$ *is crash neutral when each operation* $\mathtt{Op}_j$ *has at least one nondeterministic execution that can be masked completely by a crash and the subsequent recovery.*

$$\mathtt{pre}_j(in, \underline{x}), \underline{x}_0 = \underline{x} \vdash \langle \mathtt{Op}_j(in; \underline{x}, out)\rangle\ \mathtt{correct\text{-}recovery}(\underline{x}_0, \underline{x}, (=)) \quad \text{crash neutral run}$$

where the desired correspondence for recovering from $\underline{x}_0$ *and* $\underline{x}$ *is that the outcomes are equal* $(=)$*. When the crash alone is sufficient to mask the effect of the operation (which is often the case), this simplifies even further to*

$$\mathtt{pre}_j(in, \underline{x}), \mathtt{Cr}(\underline{x}, \underline{x}') \vdash \langle \mathtt{Op}_j(in; \underline{x}, out)\rangle\ \mathtt{Cr}(\underline{x}, \underline{x}') \tag{11.6}$$

Proof. Directly from the semantics. The precondition may be assumed because divergence of the operation is filtered out by $[\![\mathtt{Op}_j]\!]^{\blacksquare}$ in Definition 11.17. □

The latter condition (11.6) illustrates well what it means for a machine to be crash neutral: Regardless of the initial state $\underline{x}$, there is always a possibility that the operation does nothing that remains visible after a crash. For the case study, this specifically includes all operations that leave no trace on the flash storage medium, i.e., RAM operations are always masked by a crash by choosing the $\mathtt{Cr}$ predicate appropriately to model which part of the state is supposed to be in main memory.

For example, consider the MTD model of the hardware interface in Chapter 10: As discussed previously in Example 11.4, the flash hardware may sporadically refuse to write data at all. This single run is sufficient for crash neutrality. It is not required that all hardware errors are masked by a crash.

Complementarily, RAM state is characterized by setting $\mathtt{Cr}^{\mathtt{M}}(\underline{mx}, \underline{mx}') \equiv \mathtt{true}$ for an implementation level machine $\mathtt{M}^{\blacktriangleleft}$ i.e., $Cr^M = MSt_{\lightning} \times MSt$. It follows that (the $\mathtt{M}^{\blacktriangleleft}$ part) of such states are all trivially crash equivalent. For compound state $xs \oplus ms$ of $\mathtt{M}^{\blacktriangleleft}(\mathtt{X}^{\blacktriangleleft})$ we therefore have:

Proposition 11.20 (Composition of crash subsumption)**.** *When* $Cr^M = MSt_{\lightning} \times MSt$ *for a system* $\mathtt{M}^{\blacktriangleleft}(\mathtt{X}^{\blacktriangleleft})$ *then* $(xs \oplus ms) \sqsubseteq_{\lightning} (xs' \oplus ms') \iff xs \sqsubseteq_{\lightning} xs'$.

The next result is central: We can lift crash neutrality through a refinement hierarchy up to the topmost level by proving crash neutrality for every abstract specification machine. As a consequence of Lemma 11.18, we can therefore switch to the black box semantics for *every* proof of crash safe refinement.

Lemma 11.21 (Propagation of crash neutrality)**.** *If* $\mathtt{X}^{\blacksquare}$ *is crash neutral, all operations of* $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare})$ *terminate within their respective precondition, and* $\mathtt{Cr}^{\mathtt{M}}(\underline{mx}, \underline{mx}') \equiv \mathtt{true}$*, then* $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare})$ *is also crash neutral.*

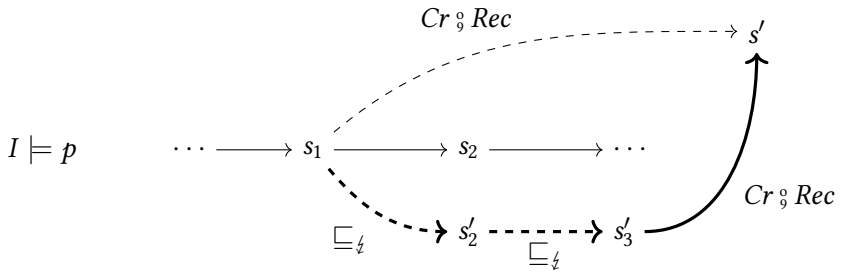


Figure 11.5: Crash subsumption for states of an execution $I = (\cdots, s_1, s_2, \cdots)$ of a program p. The dotted fat arrows denote forward propagation of the location where the crash needs to is considered, i.e., from the state s_1 of the initial execution to an alternate future s_2', s_3' which subsumes the crash in s_1. The fat arrow towards the recovered state s' the one considered by the black-box-semantics, where the dashed thin arrows amount to white box crash recovery.

As a consequence of Lemma 11.18, we have $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare}) \equiv \mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$. Note that Lemma 11.21 can potentially be extended to any combination of white/black box machines, although we'll keep the presentation to the special case we're interested in.

The argument propagates an intermediate crash in the middle of an $\mathtt{M}(\mathtt{X})$ program p towards the end of its execution over the steps of the completion as shown in Figure 11.5. Given an intermediate state $s = xs \oplus ms$ (for example s_1 in the figure) that is split disjointly into the $\mathtt{X}$ and $\mathtt{M}$ components, a subsequent step of p is either an assignment that modifies ms only, or it is a submachine call. The modification done by the assignment is masked by the crash anyway, since $\mathtt{Cr}^{\mathtt{M}}$ can result in an arbitrary $\mathtt{M}$ state: All states of $\mathtt{M}$ are crash equivalent by Proposition 11.20. Crash neutrality of the submachine permits to propagate the $\mathtt{X}$ crash over a call, justifying that it doesn't matter whether the crash happens before or after the call.

Proof. Considering each operation $\mathtt{Op} = (\mathtt{pre}, in, p, out)$ of $\mathtt{M}(\mathtt{X})$ in isolation and an arbitrary, fixed pair of states (s, s') in the white box semantics that is generated by Definition 11.7 as $I_1 \fatsemi I_2 \models p$ for a finite $I_1 = (s, \cdots, s_1)$ and some dropped remainder I_2. Existence of an alternative future computation I_2' must be shown such that $I_1 \fatsemi I_2' \models p$ and crash & recovery can be done in $I_2'.last$ instead of s_1 with the same outcome.

Furthermore, one can safely assume that I_2 is nonempty, otherwise the execution is already at the end of p and $I_2 = I_2' = (s_1)$ as the common state is collapsed by interval concatenation $I_1 \fatsemi I_2'$.

We determine an alternate successor state s_2' from the first step (s_1, s_2) of I_2 by discerning the program statement executed. The intention is that s_2' subsumes crashes in s_1. The step (s_1, s_2') then gives the first one of I_2'. The process is repeated similar to the diagonalization in the proof of Theorem 5.3 or the construction in Proposition 4.5.

The assumption that all operations of $\mathtt{M}$ terminate within their respective precondition is crucial for two reasons: On one hand, the interval I_2' is guaranteed to be finite (otherwise it does not give a final state $I_2'.last$ subsuming s'). On the other hand, each submachine call that is necessary to complete the execution of p will not give $\bot$, i.e., regardless of what happens, $\mathtt{X}$ is always called within its own precondition satisfied.

We discern the cases that determine s_2' and argue that $s_1 \sqsubseteq_{\lightning} s_2'$.

- If the first step $(s_1, s_2) = (xs_1 \oplus ms_1, xs_2 \oplus ms_2)$ of I_2 is a submachine call $\mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$, i.e., $([\![e]\!](ms_1), xs_1, xs_2, o) \in [\![\mathtt{Op}^{\mathtt{X}}]\!]$, then (11.5) provides an alternative successor xs_2' such that $xs_2' \sqsubseteq_{\lightning} xs_2$. We extend the partial run with $s_2' := (xs_2', ms_2')$ for $ms_2' := ms_1(\underline{z} \mapsto o)$ which satisfies Definition 4.14, i.e., $(s_1, s_2') \models \mathtt{Op}_j^{\mathtt{X}}(\underline{e}; \underline{xx}, \underline{z})$. Crashes in the combined state $xs_2' \oplus ms_2'$ thereby subsume the ones in $xs_1 \oplus ms_1$ by Proposition 11.20.
- Otherwise, the first transition (s_1, s_2) of I_2 leaves the state of $\mathtt{X}$ unchanged, as it can only be modified indirectly by submachine calls. We extend the partial run with $s_2' := s_2$ i.e., the next step of the program is executed. The states s_1 and s_2 are crash equivalent because the $\mathtt{M}$ crash is unrestricted and the $\mathtt{X}$ state is unchanged. □

Remark. Since p has terminating executions only, by Lemma 4.4 it is adequate to use the least fixpoint semantics $I_1 \fatsemi I_2 \models_{\downarrow} p$. However, the associated induction principle does not help with the proof: Since the proof constructs an alternative future I_2' we're leaving the original trace, specifically, prefixes of the I_2 from the original derivation mismatches the current partial extension constructed so far. This specifically means that the inductive hypothesis for q of a sequential compositions $p; q$ will not be usable, because the subinterval for q can only start with states contained in I_2 but not with a different one of the alternate execution I_2'.

The following theorem summarizes the results of this section:

Theorem 11.22 (Crash reduction for ASMs)**.** $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare}) \equiv \mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ *holds, when*

$\mathtt{Cr}^{\mathtt{M}}(\underline{mx}, \underline{mx}') \equiv \mathtt{true}$	unrestricted crash
$\mathtt{pre}_k^{\mathtt{M}}(in, \underline{xx}, \underline{mx}) \vdash \langle\!\lfloor \mathtt{Op}_k^{\mathtt{M}}(in; out) \rfloor\!\rangle\ \mathtt{true}$	termination
$\mathtt{pre}_j^{\mathtt{X}}(in, \underline{xx}), \mathtt{Cr}(\underline{xx}, \underline{xx}') \vdash \langle \mathtt{Op}_j^{\mathtt{X}}(in; \underline{xx}, out) \rangle\ \mathtt{Cr}(\underline{xx}, \underline{xx}')$	$\mathtt{X}^{\blacksquare}$ is crash neutral

Proof. From Lemmas 11.18—11.21. □

11.7 Related Work

The technology for reasoning about power cuts and more generally about system crashes caused by external events is fairly new (at the time of writing) and has been explored only recently in the context of non-trivial examples. The same idea underlies all of the approaches described below: A crash aborts the current execution in an intermediate state and is followed by a recovery operation. Some technical differences lie in the mechanism how a crash is produced as part of the semantics of programs and how recovery of nested components is addressed. In practice, the proof effort is confined by several observations regarding atomicity of the effect of a crash.

11.7.1 Modeling Power Cuts with Exceptions

Marić and Sprenger [104] observe that it is sufficient to consider crashes only when the persistent state is changed. They argue that a suitable model of the interface to the hardware either persists a write operation successfully, or the possible outcomes of a crashed write can be abstracted to an atomic but possible nondeterministic effect. A crash is modeled by throwing an exception to abort the current execution. Nested crashes and recovery are integrated into the exception mechanism in terms of handlers that are called when the

call stack is unwound. The authors observe similarly to our approach that crashes can be reduced to hardware errors, i.e., their hardware model has a similar schema as our MTD:

```
{ body } or { throw EIO }
```

Since the calculus implemented by KIV does not support exceptions, we have opted for the more direct approach using error return codes, although semantically, the approach is actually quite similar: Exceptions are propagated to their handler using special markings (see e.g. [87]) that resemble our crashed states $s_{\lightning}$.

As a downside of modeling crashes with exceptions it can be argued that the verification engineer must do this explicitly by adding exception handlers, leaving some room for introducing inadequacies, whereas in the approach presented here, adequacy depends on the semantic definitions only.

The application considered in [104] focuses on a storage manager that provides reliability through redundancy.

11.7.2 Separation Logic with Crash Conditions

The fully verified file system FSCQ for conventional magnetic disks is verified using the Crash Hoare Logic (CHL) framework [33, 34], which is explained in more detail in Chen's recent thesis [32]. CHL is an extension of Hoare Logic where each part of the program is annotated with a *crash condition* that specifies the state after a crash. CHL considers assertions for programs p of the form

$$\{P\}\ p\ \{Q\}\{S\},$$

for a precondition P, postcondition Q and crash condition S. For an uncrashed execution of p an Q must be established in the final state as usual.. For a crashed execution, all intermediate states must satisfy S, which is realized in the rule for sequential composition: the crash condition of p must imply the crash condition of a compound $p; q$ resp. that of the surrounding procedure.

Persistent and volatile state is kept disjoint: The former is modeled by a deep embedding of separation logic into the Coq prover [20], whereas the latter is represented shallowly in the logic. Crash conditions S are confined to the peristent state, which provides a canonical specification of the effect of a crash.

This reasoning effectively determines a temporal invariant $\forall\, s.(s, s') \in Cr \implies s' \models S$ that must hold in each state $s = I(k)$ of intervals $I \models p$. This mirrors the correct recovery condition (11.2) when p is taken as the concrete operation and S is the fragment dealing with the abstract level. In our simple model based on transition systems, a crash condition S could potentially be introduced into the recovery obligation in Theorem 11.3 just right before Cr^A and Cr^C to modularize the formula.

That a programs p is subject to recovery by a program r are written as $p \bowtie r$. The introduction rule

$$\frac{\{P\}\ p\ \{Q\}\{S\} \quad \{S\}\ r\ \{R\}\{S\}}{\{P\}\ p \bowtie r\ \{Q \vee R\}}$$

chains the crash condition S to the precondition of the recovery program, which establishes R in its final state and maintains the same crash condition so that it can be restarted

after crashes during recovery itself. This mirrors the embedding of crashing transition systems into regular ones in Definition 11.1.

Introduction of recovery is intended to be confined to the top-level operations of a system.[2] There is no rule to combine recovered programs in CHL in the calculus, although one can certainly implement recovery procedures in terms of several subroutines. Furthermore, attaching recovery to internal operations as suggested in [32, Figure 4-7] can be used to document or assert the relation between a part of the file system and the intended recovery subroutine, which is then called for the whole system. This coincides with Definition 11.9 of combined recovery for submachine compositon in Section 11.4. However, the difference is that we prove crash safety for submachines in isolation, whereas in FSCQ, the whole crash condition is lifted bottom-up, considering recovery only at the top-level.

CHL is based on a variant of separation logic that partitions the state into "logical address spaces", which can be seen as multiple disjoint (named) heaps that can be referred to in specifications of pre-/post- and crash conditions. CHL is mechanized in the Coq theorem prover. The authors report good results in terms of modularization and proof automation.

Similarly, Ntzik et al. [120] define a separation logic calculus for reasoning about systems/programs that exhibit crashes. The work is embedded into the views framework [45] to offload some parts of the proofs for the correctness of the calculus. Correctness of a program is written $S \vdash \{P\}\ p\ \{Q\}$, which corresponds to $\{P\}\ p\ \{Q\}\{S\}$ in the CHL approach.

The separation logic assertion P and Q describe the persistent and the durable state separately. States $s = (v, d)$ is consequently partitioned into a volatile part v and a durable part d, and the relational semantics of atomic statements a is lifted as follows

$$\begin{aligned}[\![a]\!]_{\lightning} &= [\![a]\!] \\ &\cup \{((v, d), (\lightning, d')) \mid d' \models S\} \\ &\cup \{((\lightning, d), (\lightning, d))\}\end{aligned}$$

where crashed states are marked by a special volatile part $\lightning$. The first clause reflects the normal execution of a. The second clause introduces a crashed persistent state satisfying the (given) recovery condition S. The third line propagates crashes to the end of the program text, which amounts to aborting the current execution. The precise definition of how a crash is triggered and propagated can be found in the accompanying technical report.[3]

A recoverable program is written $[p]$, where r **recovers** p defines a recovery program r for p, corresponding to $p \bowtie r$ in CHL, although the latter does not regard this as a program again. The calculus provides introduction and elimination rules for $[\_]$ around programs. It is possible to have different recovery operations for different parts of the program and recovery can be nested as stated in [120], whereas in [32] this is not permitted. Exploiting this feature in a proof amounts to making the assumption that the operating system or runtime environment maintains the stack of recovery operations somehow in a durable way.[4] This assumption is adequate for certain classes of systems, for example, services running under the supervision of the operating system.

The authors demonstrate their approach with a proof sketch of the ARIES recovery algorithm that is used in database systems. In their example, the recovery condition S is simply a (disjunctive) enumeration of all possible intermediate states. This corresponds

[2] Personal communication with the author of [32].
[3] Available at http://hdl.handle.net/10044/1/26153.
[4] Personal communication with the authors of [120].

to the nondeterministic choices that the abstract specifications in our approach make for error handling. The approach in [120] does not consider hierarchical systems, a fact that the authors acknowledge in their conclusion.

One difference between the two separation logic approaches and our formalism is that we do not presuppose the use of a particular logic to encode assertions about states. This gives more freedom to the engineer, while at the same time possible opportunities for a more streamlined proof system are given up.

Another minor difference is that the recovery condition is computed dynamically by symbolic execution in our approach and does not have to be given explicitly. In practice, we have to make sure that a corresponding abstract layer captures relevant intermediate states of the corresponding implementation anyway (as enforced by the crash neutrality condition), the question is simply whether these are given as a disjunctive formula or as additional choices of a nondeterministic program.

However, our formalization puts more emphasis on a global picture in terms of the semantics of the whole system as a set of runs. Arguably, this exposes the effect of a crash more explicitly and provides means to study the different degrees of atomicity, i.e., the white box and black box views.

The assumption that a stack of recovery operations is maintained as persistent data in [120] just reflects different view points arising by the respective case studies: goal of this work is to *provide* a robust file system, whereas Ntzik et al. [120] *assume* one and consider the application level. There is no fundamental difference, though: restriction to a single top-level recovery operation in [120] models our approach if desired; conversely, we could encode the stack of recoveries explicitly.

11.7.3 Model Checking

Koskinen and Yang [96] explore fully automatic techniques to certify existing programs as crash safe. Their correctness criterion is based on the principle that a crashed run should correspond to another normal run of the same program: *Recovered programs should not introduce behaviors that were not present in the original (uncrashed) program* [96, Section 3.1]. The benefit of this approach is that no extra work is required to e.g. annotate a given program with a specification.

The main conceptual difference to our work is that Koskinen and Yang [96] look at a single execution of a program. Upon a crash the program is simply restarted, assuming that it will try to recover its persistent data structures (the authors instrumented some of the example programs by hand to do this). While it is discussed, how repeated crashes during recovery can be handled, the paper essentially focuses on what could be compared to a single transition in the ASM model.

In [96, Section 2] the authors observe that crash-recoverability cannot be expressed in the temporal logics typically supported by the model checkers. We bypass this limitation by using the highly expressive logic RGTIL [145], which permits one to arbitrarily mix programs, weakest precondition modalities and temporal operators at the expense of automation (cf. Section 11.5.2).

The paper is based on an experimental tool called Eleven82 that is evaluated with several production quality data base systems. The promising results show that the tool produces answers as expected by the authors, in particular, several deficiencies regarding crash safety are successfully found.

Bornholt et al. [21] consider different crash consistency models from observing the crash behavior of existing file systems. They are able to check the guarantees made by existing file systems in practice using the FERRITE tool.

11.7.4 Relation to Transactions

Recovery from failure has been studied quite a bit in the context of transactions. However, we are not aware of an approach that supports a uniform way to specify unexpected crashes or that integrates into refinement.

The main difference is that we have to consider crashes at each program location, whereas rollback of transactions is triggered explicitly, see for example Hoare's work on compensable transactions [85]. Freytag et al. [62] use a specialized predicate transformer semantics to study crashes, but their approach is very specific to databases and too restricted for our purpose.

11.7.5 Other Approaches

In [148] a high-level modeling language specifically designed for file systems is described. Examples cover sophisticated optimizations such as reordering of writes and versioning. However, the modeling language is not intended for an actual implementation of a file system. The optimizations that can be described with this language are beyond the scope of this thesis.

The temporal logic conditions of Section 11.5.2 can be verified for example in Dynamic Trace Logic [23].

The interrupt operator of CSP is similar to our semantic definition of a crash regarding the possibility to abort a running operation: Process $P \triangle_i Q$ denotes that P executes until an (external) event i occurs, after which P is discarded and Q is started. The operator originates from [84] and is further explored in [106]. In [134], a trace-based semantics is given that is equivalent to our white-box semantics of programs $[\![p]\!]^{\nearrow}$.

Chapter 12

Dealing with Power Cuts

Summary. The concern of power cut safety spans the whole refinement tower: every layer must deal with it. For the overall system's ability to recover from such events, it is crucial that the measures implemented at each level work together well. This chapter details some of the intricate aspects that arise in this regard and outlines the challenges towards a coherent, working approach.

Publications: This chapter is based on [54, 123].

Contents

This chapter describes how crash safety is realized by the cooperation of the components presented in the Chapters 7 to 10. Each layer addresses a specific aspect of power cut safety that is naturally associated with the concepts it realizes. The general pattern considers refinements $\mathtt{A}^{\blacksquare} \sqsubseteq \mathtt{C}^{\square}(\mathtt{X}^{\blacksquare})$, where $\mathtt{C}^{\square}(\mathtt{X}^{\blacksquare})$ is an implementation level machine which takes the white box crash semantics as explained in Example 11.5. The specification $\mathtt{A}^{\blacksquare}$ is atomic and therefore takes the black box semantics as explained in Example 11.4 and so is the submachine $\mathtt{X}^{\blacksquare}$ as an abstract specification of the subcomponent used by $\mathtt{C}$.

A systematic description of the models is given by instantiating the refinement theory for crash safety of Chapter 11, i.e., by defining the crash predicates and recovery operations. For a machine $\mathtt{M}^{\square}$ with state $\underline{x}$, we write

crash (M) $\varphi(\underline{x}, \underline{x}')$

to define the crash transitions $\mathtt{Cr}^{\mathtt{M}}$ from unprimed to primed states by a formula φ. The recovery operations $\mathtt{Op}^{\mathtt{M}}_{\mathrm{rec}}$ are introduced analogously to regular ones with the name `recovery` prefixed by the respective machine

M_recovery(; *out*, *err*)

body

In addition to the description of the models, it will also be outlined how submachines $\mathtt{X}^{\blacksquare}$ are proved to be crash neutral, so that we can switch to the black box semantics for their supermachines, i.e., to exploit that $\mathtt{C}^{\square}(\mathtt{X}^{\blacksquare}) \equiv \mathtt{C}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ by Theorem 11.22.

The remainder of this chapter is structured as before. A brief technical overview will be given for the benefit of the big picture, before the problems and solutions for crash tolerance on the individual layers are explained.

12.1 Summary and Technical Rundown

Similarly to the presentation of the models and refinements for the functionality, this section gives an overview of the different effects of power cuts on each layer and the respective approaches to deal with these.

The specification of crash behavior of the formal POSIX model from Section 7.6 is briefly repeated. A crash leaves the directory tree *tree* and the file store *fs* untouched but destroys the registry of open files *oh*, i.e., oh' is arbitrary. The recovery operation reinitializes *oh* and deletes all orphaned files, i.e., the ones that were already unlinked from the tree but were still referenced by open file handles at the time of a crash:

crash (POSIX) $\quad tree' = tree \wedge fs' = fs$

```
posix_recover( )
  oh := ∅
  fs := fs \ orphans(tree, fs)
```

The requirement induced is twofold: The implementation must work hard to achieve atomicity of operations, because the model POSIX$^{\blacksquare}$ is a black box machine, and the implementation must be able to determine the set of orphaned files and maintain it correctly.

In the implementation, two focal points determine the global strategy for power cut safety: 1. redundancy of the RAM index and 2. transactional writes. Aspects that are somewhat disconnected from the major strategy but equally important are free space management and garbage collection in the journal layer, and incremental commit of the B^+ tree, which are both complicated by power cuts.

It should be commented that this technical rundown is not presented alongside the models of the refinement hierarchy but instead by the concepts that need to be addressed. This stems from the fact that power cut safety is a cross cutting concern involving multiple layers to address a particular aspect.

Redundancy of the RAM Index and Orphaned Files. Within the flash file system core, the major concern is to demonstrate redundancy of the RAM index, i.e., that it can be recovered *fully* from the outdated version on flash by applying all the modifications found in the log as expressed by the recovery invariant (9.2) in Section 9.4.

However, the presence of orphans complicates the matter. In the implementation, these are recorded explicitly in two sets *ro*, *fi*: `Set⟨Key⟩` (RAM orphans and flash orphans), containing the keys of the inodes that correspond to files without hard links in the RAM index *ri* resp. the flash index *fi*. The recovery invariant must be adapted to remove these.

invariant (FFS) $\quad ri \setminus\!\setminus ro = \texttt{replay}(log, fi \setminus\!\setminus fo, fs), \qquad (12.1)$

i.e., the resulting ram index is the current one stripped of all obsolete files. Recovery starts by removing those orphans that had been committed with the flash index. The double backslash $\setminus\!\setminus$ signifies that it is not sufficient to merely remove the inode keys stored in *ro* resp. *fo* but also the keys of all of their data nodes storing the VFS pages of the files' contents. Hence, power cut recovery is directly linked to maintaining orphaned files. Although there are algebraic specifications for $\setminus\!\setminus$ and `recover` to simplify the reasoning, these must be realized by ASM operations in the implementation, too.

The practical benefit of pushing this property into the file system core is that a crash doesn't leak through the interface. In other words, all the complexity of handling power

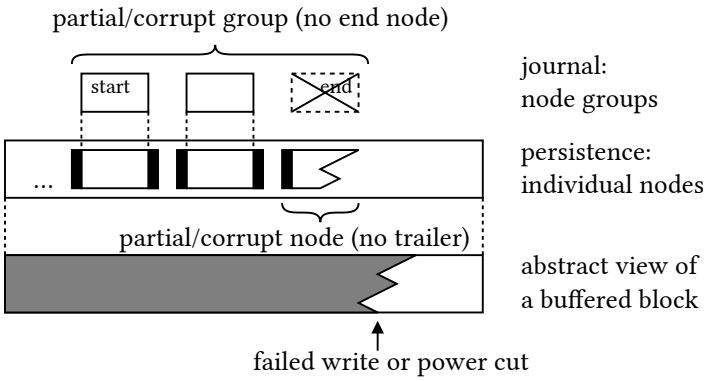


Figure 12.1: Detecting partially written nodes and groups.

cuts properly is pushed down into the FFS core and lower layers, specifically, the VFS component that calls the FFS at runtime does not have to worry at all about crashes. In the formal development this manifests in a trivial crash predicate for AFS which specifies the FFS core: the state remains unaffected apart from deleting the orphaned files as required by the top-level POSIX model.

Transactional Writes. The flash file system core relies on the capability of its journal subcomponent to write multiple nodes within transactions that take effect atomically wrt. hardware errors and power cuts, as specified in Section 9.2. This feature is realized jointly by the implementation of the journal and the persistence layer. The former is responsible for *grouping* nodes, whereas the latter is responsible to making *individual* node-writes atomic.

Grouping is achieved by start resp. end flags that are stored in the nodes on flash. The first node of a group has the start flag set, whereas the last one has the end flag set. A singleton group, for example, sets both flags. With this mechanism, it is possible to detect groups of nodes that are not complete.

Atomicity at the level of individual nodes is achieved somewhat similarly. The on-flash encoding of nodes surrounds the actual data by a header and a trailer. It can be recognized whether the trailer of a node has been persisted completely, and only then the node makes a contribution to the file system's state. Otherwise, the node is discarded, potentially leading to a partial group as an effect.

The mapping of the data structures of the two layers is depicted in Figure 12.1. At the bottom lies a logical erase block that stores a contiguous sequence of bytes. In the middle, three group nodes as stored by the journal are shown, that are abstracted to their payload in the journal at the top. In the figure, a power cut has happened so that some of the bytes could not be transferred to the storage medium. The jagged border of the grey part that has been written signifies that it is unknown how many bytes have been persisted. In the persistence layer this causes a corrupt trailer of the last node, which is then dropped. The journal detects that this group was partially written, which is then correctly discarded to undo a top-level operation.

This chapter will not detail the model, which has been published in [54] including additional concepts such as garbage collection.

12.2 High-Level Crash Recovery

This section describes the effect of a power cut at the level of VFS and AFS, specifically, how the requirement imposed by POSIX is realized at that level. The specification is actually trivial. A crash leaves the AFS data structures untouched but erases knowledge about the open file handles *oh* which are stored in RAM. This mirrors the POSIX counterpart shown above.

crash (VFS(AFS)) $\quad dirs' = dirs \wedge files' = files \quad$ (and oh' arbitrary)

The recovery operation simply discards the orphaned files and reinitializes *oh*:

```
afs_recover( )
  dirs := dirs \ orphans(dirs, files)
  files := files \ orphans(dirs, files)
```

```
vfs_recover( )
  afs_recover( )
  oh := ∅
```

The set of orphans selects those inode numbers that have no links pointing to them:

$$\texttt{orphans}(dirs, files) := \{ino \in \texttt{dom}(dirs, files) \mid \texttt{links}(ino, dirs) = \varnothing\}$$

Note that since the eviction protocol of Section 8.4 applies to directories too, there may temporarily be directories that are unlinked from the tree, and consequentially these have to be removed as well.

This can immediately be mapped to POSIX provided that determining the number of links coincides: For directories, the abstraction

$$\texttt{dirs}(\texttt{ROOT\_INO}, tree)(dirs') \quad \text{for } dirs' = dirs \setminus \texttt{orphans}(dirs, files)$$

is upheld by the fact that `dirs` uniquely determines the domain of the directory store, more specifically, it gives a tight upper bound for the directories: a directory that is no longer reachable from the root of the tree must not be allocated in $dirs'$. For files, deletion distributes through the abstraction relation `files` (cf. Section 7.6 and 8.8):

$$\texttt{files}(fs \setminus \texttt{orphans}(tree, fs)) = \texttt{files}(fs) \setminus \texttt{orphans}(dirs, files).$$

The study of crash and recovery is completed by an argument why the operations of AFS are crash neutral, so that the above conditions are sufficient by Theorem 11.22. For most operations this is trivial due to the nondeterministic error handling by pattern (6.2): such AFS operation can fail without changing the state.

However, there is one operation that does not fit this pattern, namely $\texttt{afs\_evict}(ino)$ (shown in Figure 8.5), which takes a single inode number *ino* and deletes it from the respective store *dirs* or *files* provided it has no more links. The reason why `afs_evict` is special is that it must not fail as discussed in Section 8.4: The operation that had unlinked the file or directory denoted by *ino* has already taken effect. We therefore need the full generality of Lemma 11.19, which permits to mask the effect of `afs_evict` by a subsequent crash and recovery.

To prove that $\texttt{afs\_evict}(ino)$ is crash-neutral when $\texttt{links}(ino, dirs) = \varnothing$, after instantiating Lemma 11.19 to the respective crash predicate and operation, it remains to be shown that

$$files \setminus \texttt{orphans}(dirs, files) \subseteq files \setminus \{ino\} \setminus \texttt{orphans}(dirs, files),$$

i.e., deleting this particular *ino* by `afs_evict` subsumes the crash before its invocation. This holds since $\texttt{links}(ino, dirs) = \varnothing$ implies $ino \in \texttt{orphans}(dirs, files)$.

Before we progress to the more complex lower layers it should be emphasized that integration of power cut safety at this level of abstraction was fairly easy, which can be taken as an indication that the choice of abstraction is right: It achieves to split off the effort from the complex VFS implementation; the reason is of course that AFS does not discern between persistent and volatile state.

12.3 Recovery in the Flash File System

We first describe how recovery is realized in the flash file system core when orphans are disregarded. In the event of a crash, the RAM state is lost, whereas the persistent state consisting of the flash store, the log, and the flash index is kept (cf. Figure 9.1):

$$\textbf{crash}\ (\text{FFS}) \qquad fs' = fs \wedge log' = log \wedge fi' = fi \tag{12.2}$$

The recovery invariant states that this is not a problem. It is always possible to reconstruct the RAM index from the other three data structures:

$$\textbf{invariant}\ (\text{FFS}) \quad ri = \texttt{replay}(log, fi, fs).$$

As `replay` is an algebraic function, it cannot serve as an implementation of the recovery operation in the flash file system. Recovery is therefore given by a procedure `fs_recover()` (not shown) with the same effect and we prove the correspondence

$$\vdash \langle\!|\, \texttt{fs\_recover}(fs, log, fi; ri) \,|\!\rangle\ ri = \texttt{replay}(log, fi, fs).$$

The algebraic function `replay` is specified recursively over the *log* by two cases

$$\begin{aligned} &\texttt{replay}([\,], fi, fs) \coloneqq fi \\ &\texttt{replay}(adr + log, ri, fs) \coloneqq \texttt{replay}(log, \texttt{replayone}(adr, fs[adr], ri), fs) \end{aligned} \tag{12.3}$$

For an empty log, the result is simply the flash index *fi*. Otherwise, the node $fs[adr]$ at the first address in the log is inspected by an operation `replayone` and integrated as necessary into the RAM index *ri* built up so far and the replay recurses.

12.3.1 Incremental Justification of Recovery

The approach due to Schierl et al. [147] to maintain the recovery invariant will be shown briefly. It is a prerequisite to understanding how orphans are maintained in the subsequent section.

An immediate consequence of (12.3) is that `replay` distributes over the catenation of two log lists log_1 ++ log_2, where replaying the second log is started in the RAM index that is the result of applying the entries of the first list:

$$\texttt{replay}(log_1 \texttt{++}\ log_2, fi, fs) = \texttt{replay}(log_2, \texttt{replay}(log_1, fi, fs), fs) \tag{12.4}$$

This means that one can reason about how subparts of the log are recovered, which provides a lever to maintain the recovery invariant incrementally. For instance, the FFS operations `fs_create` and `fs_unlink` as shown in Figure 9.3 and Figure 9.4 append three

entries to the log, one for the parent directory, one for the file to be created or unlinked, and one for the directory entry, resulting in three new addresses adr_1, adr_2, and adr_3 where the new nodes have been stored, i.e., the new log will be $log' \coloneqq log \mathbin{+\!\!+} [adr_1, adr_2, adr_3]$ for three updates to the flash store $fs' \coloneqq fs[adr_1 \mapsto nd_1, adr_2 \mapsto nd_2, adr_2 \mapsto nd_2]$. Using (12.4) to unwind the log backwards from the end, we just have to make sure that `replayone` coincides with the updates to the RAM index to store these new addresses. For instance, replaying an inode discerns whether the link count is zero, in which case the file is to be removed. Otherwise, the index is updated with the key from the node (cf. (9.6)) to the address found in the log:

$$\texttt{replayone}(adr, \texttt{inodenode}(key, \cdots, nlink, \cdots), ri) \coloneqq \tag{12.5}$$
$$\begin{cases} ri[key \mapsto adr] & nlink \neq 0 \\ ri - key & nlink = 0 \end{cases}$$

Instantiating (12.4) with $log_1 \coloneqq log$, $fs \coloneqq fs'$, and $log_2 \coloneqq [adr_1, adr_2, adr_3]$ shifts the *new* entries to the outer `recovery`, which can then be broken down by its recursive definition.

The observant reader may have noticed that (12.4) works for taking the same fs twice on the right hand side of the equation. We have $ri = \texttt{replay}(log_1, fi, fs)$ from the old invariant but we need $ri = \texttt{replay}(log_1, fi, fs')$ for the bigger fs' by unfolding (12.4) for the new state. However, it is easy to see from (12.3) that `replay` is robust against extensions of fs by *fresh* addresses not contained in the log.

We remark on two aspects.

- Maintaining the recovery invariant in a non-incremental fashion is not feasible. The strategy crucially depends on the availability of a concrete, finite log_2 for which `replay` can be compared to the outcome of the concrete operation that had produced the entries stored in log_2. The trick employed here is to transform the proof obligation correct recovery (Proposition 11.14) into an invariant.
- It is also important that the groups of addresses are written atomically. Replaying for instance the partial group $[adr_1, adr_2]$ where the last address is left out would lead to an inconsistent state (for `fs_create` we have a dangling directory entry when the target file does not exist).

Crash neutrality of the journal layer Section 9.2 is trivial: all operations, including `commit`, follow the error specification pattern (6.2) and can fail without any effect. The index machine is also crash neutral: It modifies the memory data structure ri only which is erased by the crash (12.2).

12.3.2 Maintaining Orphaned Files

We have already seen in the Sections 7.6 and 8.4 that not all files must be referenced by the directory tree any more, as it is possible to only refer to a file's content via an open file handle. In the abstract world, these are easy to determine just by collecting them from the state, and one can specify them for the file system's core as well (as a set of inode keys):

$$\texttt{orphans}(ri, fs) = \{\texttt{inodekey}(ino) \in ri \mid fs[ri[\texttt{inodekey}(ino)]].\texttt{nlink} = 0\}, \tag{12.6}$$

This implicitly assumes that the `nlink` field stores the correct value, namely the number of hard-links to that particular file identified by *ino*. However, computing this set on-the-fly

is infeasible: the implementation would have to traverse the *entire* index, which in general may be large and therefore such a scan could be quite time-consuming.

The solution, of course, is to store and update the set explicitly. There must be two versions: one that lives in main memory and matches the RAM index (RAM orphans), and one that is written to flash and matches the outdated flash index (flash orphans). These are given by two state variables

state vars (FFS) $ro\colon \mathit{Set}\langle \mathit{Key}\rangle$
spec vars (journal) $fo\colon \mathit{Set}\langle \mathit{Key}\rangle$

While the version *ro* in RAM is made an implementation level data structure of the final system, the on-flash version *fo* is still kept as auxiliary state. The reason is that the latter must somehow be encoded down to the bytes of the hardware, and it is therefore maintained by the lower layers of the refinement hierarchy. The corresponding invariants state that the two variables reflect the correct sets respectively:

invariant (FFS) $ro = \texttt{orphans}(ri, fs)$
invariant (journal) $fo = \texttt{orphans}(fi, fs)$

The second invariant is trivially established from the first at the time of the commit, which sets $fi := ri$ and $fo := ro$. The first one is more intricate: As *ro* is updated alongside the index *ri* by normal operations, one has to connect the `nlink` field of the nodes (referred to by (12.6)) to the allocation and deallocation of the directory entries (not shown).

Crash specification (12.2) of the journal is extended to preserve the set of flash orphans

crash (index/journal) $\cdots \wedge fo' = fo$

The operations `unlink`, `rmdir`, `close`, and `rename` must be adapted to update the set *ro* when the last reference to an inode is dropped, i.e., they now include code of the form

if $nd.\texttt{nlink} = 0$ **then** $ro := ro \cup \{key\}$

In the presence of orphans, the full recovery invariant reads as

invariant (FFS) $ri \setminus\setminus ro = \texttt{replay}(log, fi \setminus\setminus fo, fs),$

Let's try to re-establish the proof principle of Section 12.3 based on the simple splitting principle (12.4). We consider the simple case of an operation that writes a single node *nd* identified by *key* to a (fresh) address *adr*. For the sake of the argument, assume that this node corresponds to a freshly orphaned file. Let the primed versions

$$ri' = ri[key \mapsto adr] \qquad fs' = fs[adr \mapsto nd]$$
$$ro' = ro \cup \{key\} \qquad log' = log \mathbin{+\!\!+} [adr]$$

denote the resulting state. The RAM modifications are shown on the left, the flash modifications are shown on the right. The flash index and orphans are unmodified. The (extended) recovery invariant as shown above is known for the starting state.

We can derive

$$
\begin{aligned}
&\texttt{replay}(log', fi \setminus\!\setminus fo, fs') \\
&\quad = \texttt{replay}(log \texttt{ ++ } [adr], fi \setminus\!\setminus fo, fs') && \text{(by assumption)} \\
&\quad = \texttt{replayone}(adr, nd, \texttt{replay}(log, fi \setminus\!\setminus fo, fs)) && \text{(by (12.4) and narrowing down } fs\text{)} \\
&\quad = \texttt{replayone}(adr, nd, ri \setminus\!\setminus ro) && \text{(recovery invariant)} \\
&\quad \stackrel{!}{=} ri' \setminus\!\setminus ro' && \text{(to be shown)}
\end{aligned}
$$

where the last step represents the commutation that falls out and remains to be shown. Since we have assumed that the inode referenced by *key* is an orphan (i.e., $nd.\texttt{nlink} = 0$), we can further narrow this down by definition (12.5) of `replayone`:

$$(ri \setminus\!\setminus ro)[key \mapsto adr] - key \stackrel{!}{=} ri \setminus\!\setminus (ro \cup \{key\})$$

which holds.

In general, this principle boils down to a lemma of the following form:

$$\texttt{replay}(log, ri \setminus\!\setminus ro, fs) = \texttt{replay}(log, ri, fs) \setminus\!\setminus ro$$

12.3.3 Additional Invariants

In practice, a number of side conditions must be established so that this holds. These must be maintained as part of the invariants of the system:

- Each address encountered in the *log* must point to a valid node in the flash store, i.e., $\texttt{dom}(log) \subseteq \texttt{dom}(fs)$.
- The types of the keys stored in the nodes $fs[adr]$ for addresses *adr* in the log must match the type of the nodes.
- Obviously, the sets of orphans *ro* and *fo* may only contain inode keys, and the root inode is not one of them.
- Each address must not be in the RAM index already, i.e., $\texttt{replayone}(adr, nd, ri)$ has the precondition $adr \notin ri$ for the *intermediate* index that is partially recovered.

The last requirement is actually not straight forward to specify The solution is to capture all of these requirements by a predicate `replaysafe` that mirrors the recursion of `replaylog` and checks whether the respective conditions are satisfied before replaying a particular entry with the RAM index rebuilt so far. The predicate `replaysafe` is maintained incrementally just like `replaylog`.

Furthermore, the operation `index_newino` shown in Section 9.2 to determine an unused inode number must be prevented to reuse those in the set of *flash orphans*. This requirement was not obvious. Note that this can be achieved by *never* reusing inode numbers, though.

12.4 Related Work

The specification of recovery of the log in the file system core has been addressed in related work by [122], which is in fact based on our model [147]. Their goal is to explore points-free relational modeling and proofs. They model a simplified version of recovery that does

not consider orphans, and restricts the log to the deletion of files only (but not creation). This bypasses many of the problems outlined in Section 12.3, as it becomes possible to state recovery in a closed, non-recursive form. The recovery just deletes all entries in the log, as dependencies between operations recorded in the sequential log are nonexistent.

In comparison to UBIFS, we have not employed multiple journals, i.e. separate sequential logs for different kinds of data. UBIFS separates inodes and directory entries from data pages as they have different access patterns. While the former two types of nodes must be written mostly synchronously, data pages are cached (by the Linux kernel) for some time before they are written to disk. Multiple journals come with the downside that recovering the sequential ordering of the nodes during recovery is complex and requires additional concepts such as sequence numbers.

Chapter 13

Summary and Discussion

Summary. This section and discusses the results of this thesis and the Flashix project. The theoretical results and their influence on the modeling and verification are summarized. Statistics about the development effort are presented and we outline the lessons learned from the project.

Contents

13.1 Theoretical Results

We have defined a refinement theory for data type ASMs with submachines (Chapter 49, which respect information hiding, and can recover from power failures. The theory has been a key to enable modular and incremental development of the flash file system case study. The theory links explicit observations which permit to reason about the systems' evolution and behavior in time to compositional reasoning to enable a modular development. The theoretical contribution is thereby a definition of refinement $\mathtt{A} \sqsubseteq \mathtt{C}$ between systems that permits to replace the abstract specification A by its concrete implementation C within any context C, as expressed by the first central result (Chapter 5):

Theorem 5.9 (Compositionality).
$\mathtt{A} \sqsubseteq \mathtt{C} \implies \mathtt{M}(\mathtt{A}) \sqsubseteq \mathtt{M}(\mathtt{C})$.

The technical achievement behind this theorem is that it is based on the *expressive* Abstract State Machine formalism. As a consequence, many concerns needed to be addressed, such as potential divergence of operations, correct handling of preconditions, and most importantly submachine calls. The proofs for Theorem 5.9 explain *how* observations are threaded through the computation alongside the interactions between the submachine and its context. The outcome is a system model that is fully recursive: submachines and contexts are defined using the same formalism.

The theory is extended by explicit modeling constructs to specify the effect of power cuts and other system crashes that are triggered by events not under the control of the program (Chapter 11). It is shown how verification of crash safety can be integrated seamlessly with a refinement based approach, preserving its nice aspects of incremental and modular developments. Moreover, on the technical side, the introduction of power cuts and crashes does not affect the proofs for functional correctness, i.e., regular (uncrashed)

executions of the system are preserved by the theory so that the well known principle of forward simulation applies.

Central to the theory are two views of the crash behavior of a system. The white box crashing semantics of machines $\mathtt{M}^{\square}$ captures how a system is affected by a power cut in practice. This semantics is thereby adequate to study implementation level components which will eventually run as part of the final flash file system. It is contrasted by the black box crashing semantics $\mathtt{M}^{\blacksquare}$ that abstracts the degree of atomicity where crashes are triggered. This view justifies to treat the operations specification level machines atomically. Three composition patterns are discussed that naturally arise from the different system compositions in a refinement chain and the substitution of abstract machines by their concrete counterpart at runtime. Purely white box machines $\mathtt{M}^{\square}(\mathtt{X}^{\square})$ correspond to the running code. A white box context with an abstract black box submachine $\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare})$ corresponds to the machines in the refinement hierarchy. Purely black box machines $\mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ are independent of the intermediate states of a computation and consider crashes in between completed operations only. These are of particular interest because standard weakest-precondition calculus can be used to reason about their properties. For the formulation of the different patterns it was crucial that there is a well defined central place to consider the degree of atomicity of systems component at the operations of their interface.

It is demonstrated that the compositionality result previously proved for regular machines transfers to the crash semantics without much need to adapt the proof. This is summarized by the second result:

Theorem 11.10 (Compositionality in the presence of crashes).
Let $\mathtt{C} \in \{\mathtt{C}^{\square}, \mathtt{C}^{\blacksquare}\}$ *and* $\mathtt{A} \in \{\mathtt{A}^{\square}, \mathtt{A}^{\blacksquare}\}$,
then $\mathtt{A} \sqsubseteq \mathtt{C}$ *implies* $\mathtt{M}^{\square}(\mathtt{A}) \sqsubseteq \mathtt{M}^{\square}(\mathtt{C})$ *and* $\mathtt{M}^{\blacksquare}(\mathtt{A}) \sqsubseteq \mathtt{M}^{\blacksquare}(\mathtt{C})$.

While white box machines $\mathtt{M}^{\square}$ are adequate to describe implementation level machines, there is a high effort associated with the corresponding refinement proofs. The reason is that there is a high number of intermediate states. It is desired to reduce the verification effort to those states which are actually relevant to the study of power cuts. A direct observation is that transitions that affect the in-memory state only are irrelevant. This notion has been formalized by a definition of subsumption of crash behavior.

With *abstract* models of the refinement hierarchy, however, the partitioning into volatile and persistent state is less clear. Key to the generalization of this observation was the introduction of the formal criterion of *crash neutrality*, which provides a means to study the subsumption of crashes for specification level machines. The result of this approach is the third result that reduces the states that are relevant for power cut analysis significantly:

Theorem 11.22 (Crash reduction).
$\mathtt{M}^{\square}(\mathtt{X}^{\blacksquare}) \equiv \mathtt{M}^{\blacksquare}(\mathtt{X}^{\blacksquare})$ *if* $\mathtt{X}$ *is crash-neutral and* $\mathtt{M}$ *has RAM state only.*

The technical achievement behind this result propagates crash neutrality from the submachine to its context by *simple* conditions. As all of the models in the Flashix file system are crash neutral by construction, we were able to conduct the verification using standard weakest precondition verification technology.

Summary. The methodology presented in this thesis gives a uniform way to specify functional correctness as well as crash tolerance of systems by *refinement*. Proof principles for the general case and several specialized cases are derived for the verification. Central to the theory is compositionality, which permits one to break down the development into individual pieces, so that complexity can be introduced incrementally and modularly.

13.2 Practical Results

The practical contribution of this thesis lead to the development of a running file system for flash memory, which is functionally correct, power cut safe. It moreover implements modern strategies to efficiently deal with the characteristics of the underlying storage hardware, which differs significantly from traditional magnetic disks.

There is a large gap between the representation of a POSIX compliant file system as abstract tree and the low level representation in terms of the blocks and bytes of flash memory. A number of concepts were modeled abstractly in this thesis as summarized below. Working out separation of concerns so that they can be captured abstractly potentially clarifies how existing file system implementations work should work internally.

A formal model of the POSIX interface for file system operations describes the expected behavior of compliant file systems (Chapter 7). It is on one hand highly abstract, which is reflected in its concise presentation. On the other hand, it does not give in to conceptual simplifications. Instead, it tries to be as faithful to the POSIX standard as possible, which results for instance in file access through handles, the support for hard-links, orphaned files, and proper handling of errors. As the Flashix file system adheres to this established specification, it can be integrated directly into the existing software landscape, providing a verified and better structured alternative to existing solutions such as UBIFS.

Based on the modular refinement theory, this thesis demonstrates how an implementation of the POSIX model can be developed so that all generic concerns can be separated from implementation specific concepts (Chapter 8). The result is a modular architecture that simplifies the specification and proofs. The decoupling of the system into the Virtual File System (VFS) and the Abstract File System (AFS) specification lends itself to reuse of the components. Moreover, since the interface between VFS and AFS reflects the corresponding counterpart of Linux, it becomes possible to take this as a starting point for other approaches: Related work [8] takes up the AFS model for the specification of BilbyFS, which runs as part of the Linux kernel.

At the heart of Flashix is the Flash File System core component that realizes the main strategies to deal with the characteristics of flash memory (Chapter 9). It is based on a high-level and abstract model of the central data structures, namely the index and the journal and ties them together to provide a file system implementation that adheres to the AFS specification.

Summary. The presented models of the Flashix file system contribute to the development of verified software systems, by describing complex file system concepts abstractly and by ultimately providing a runnable implementation that can be used as a storage solution today. This result underpins the claim that the chosen development approach is *practical* and scales to large and realistic problem sizes.

13.3 Statistics and Development Effort

To get an impression of the size of the Flashix project, some statistical data is given. The overall development done in KIV comprises of 18 layers as shown in Figure 2.2, of which eight are compiled into the running code, whereas the remaining 10 layers are abstract ones. At the time of writing there are approximately 15 thousand lines (kLoC) of specification code, which is partitioned into setting up the algebraic data types (about a third), the

remaining larger part is ASM code. The generated Scala code consists of roughly 7 kLoC. It corresponds almost directly to the KIV implementation models. The C code is in the order of 13kLoC as it needs to be augmented by explicit memory management and an encoding of algebraic data types as C data structures.

The integration of the C code with the FUSE library takes approximately 700 lines of code. In addition three standard tools are provided, namely `mkfs` (90 lines) to format an MTD device with the base layout of Flashix, `mount` (145 lines), which hooks up the file system into the directory structure of Linux, and `fsck` which currently dumps some information and statistics of the on-flash data structures (the tool is currently not capable to check consistency as the name would suggest). The small size of these tools stems from the fact that they are just frontends for the generated code. For example, the `mkfs` utility calls a `format` operation that is part of the formal development.

In total, 9 refinements have been verified, leading to some 3000 theorems and helper lemmas to automate the verification and to prove difficult facts about the correspondence of the models (roughly half of these by the author).

The overall net effort of the development is about 6 person years. For the specific models presented in this thesis, we estimate that it took two months setting up the POSIX specification, which includes the time to understand the requirements imposed by the standard and the time to settle for a design that is abstract enough but does not leave out essential concepts. A similar effort was spent in developing the initial versions of the VFS model and the AFS specification (where the latter could be based on the existing model [147] of the flash file system). However, as explained below, these numbers do not reflect the continuous maintenance of the models in response to changes made necessary by integration with the rest of the refinement hierarchy.

Initial development of the core model of the flash file system took about six month. Adapting the existing specifications to the layers above and below was a significant effort that is difficult to measure separately.

Conducting the proofs presented here took approximately two months for the refinement of POSIX to VFS. The initial proof of functional correctness for the flash file system core was straight forward, except for the interplay between the deletion of files and hard links.

13.4 Lessons Learned

In our opinion, the file system challenge is interesting for the reason that a wide conceptual gap must be bridged. Abstractly, the file system is described as a tree indexed by paths, whereas the hardware interface is based on erase blocks and bytes. The strategies we have taken from the Linux VFS and UBIFS to map between these interfaces deal with many different concepts.

Starting Middle Out and Throw-Away Models

Development of the Flashix file system has started out with the central model [147] of the core concepts. This model has served as an anchor-point to incrementally develop the rest of the model hierarchy. Thus, we used a middle-out approach. Subsequent development took place at the level of POSIX and MTD to determine the boundary interfaces of the system. The rest of the development followed incrementally.

On several occasions we have introduced models into the refinement hierarchy to clar-

ify concepts but later on decided not to keep these. For example, the additional invariants in Section 12.3.3 for the flash file system core that are needed for correct recovery in the presence of orphans were determined by an abstract reformulation of the problem. We have extended the AFS model duplicating its state with one version roughly corresponding to the outdated flash index, linked by a list of log entries to the two stores *dirs* and *files* (AFS + recovery in Figure 2.2). Hence, it was possible to study the problem at a high of abstraction and with a representation of state that is much more amenable to formal proofs. After these invariants had been determined, they could be restated at the level of the flash file system (it was unfortunately not possible to transfer the results directly in terms of the refinement).

At the end, these intermediate models were not kept because they introduce additional maintenance effort, see the comments on change management below.

The Need for Abstraction

Capturing the different concepts that are relevant in the development of a verified file system at the right degree of abstraction proved to be a major design challenge. Not all of these concepts have a direct counterpart in the implementation level models. For example, the abstract log of the flash file system model in Chapter 9 is represented implicitly in the implementation only.

When designing a component that lies somewhere in the middle of the refinement hierarchy, there are two conflicting goals:

Abstraction: From the client's perspective, the interface should be easy to use. In a formal context it means that not only the should operations capture the problem domain well and map to the conceptual operations expected from the interface, but also that the formal model of the interface should be as abstract as possible to facilitate reasoning and proofs about *using* the interface.

Realization: Conversely, when an interface abstracts away many technical details of the implementation, the conceptual gap to the implementation becomes large. It may be challenging to *satisfy* the requirements expressed by its specification, e.g. elaborate code for error recovery to mask irregularities during operation.

The sweet spot lies in interfaces that capture as much of the problem domain as possible without leaking implementation details. This means that in particular the state space of specifications should be *simple* but *complete* in the sense that potential redundancy is already introduced as auxiliary state. An instance of such auxiliary state is shown in Section 8.2. Abstract files and directories explicitly store the number of hard-links in the AFS model (field `nlink`) in and similarly the number of entries (`size`) and subdirectory (`nsubdirs`) of a directory. This permits one to prove on the abstract level invariants for these as shown in Section 8.7. The refinement proof that links AFS to its implementation makes these invariants available on the concrete level as well, see for instance [140].

Prerequisite to defining suitably abstract models are strong abstraction capabilities of the used tools, which has been observed before by Baumann et al. [17]. KIV supports arbitrary user-defined data types (given suitable axioms), which was for example exploited to abstract the pointer structure to an algebraic tree and to abstract the sparse pages of files to streams (see Section 8.8). Capturing the file identifiers referenced from the POSIX directory tree as multisets instead of sets (Section 7.5) is another example where the freedom to choose an appropriate data structure was helpful.

Sometimes we could have taken benefit from a stronger typing system in the logic of

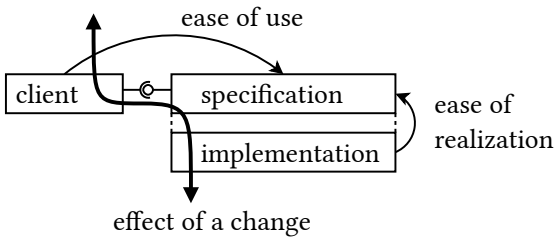


Figure 13.1: Conflicting design goals for the specification of intermediate interfaces, balancing concerns of use and implementation. Changes to requirements and models tend to propagate through the hierarchy in both directions.

the tool, such as predicative subtypes [136]. With predicative types small invariants can often be expressed in a concise way as part of the type (e.g., arrays of a fixed length), reducing for example the number of explicit preconditions and assumptions for lemmas. Furthermore, proofs required for correct type checking can be separated from the main argument, which potentially simplifies the latter.

Dealing with Change

Finding a good specification of an intermediate interface is hard in the sense that design decisions and in particular changes affect the layers above *and* below a given intermediate layer as shown Figure 13.1. To satisfy both the client and the implementer of such an interface, compromises may be involved and in the light of new insights about the problem such interfaces evolve. Consequences of changes to a component and its specification tend to propagate through the refinement hierarchy both upwards and downwards and other layers have to be adapted to cope (denoted by the fat arrow in the figure).

There is a fine balance between the possibility to satisfy the contract of an interface at all and to make its use feasible and one can never get it right in the first place. Once the general decomposition of a system has been worked out the task of fine tuning and aligning all of the different parts becomes the dominating development effort.

The major source of changes stems from error handling strategies and dealing properly with power cuts. We estimate that between one third and half of the effort of the project can be related to just these two aspects.

Here the elaborate dependency management provided by the KIV system (see e.g. [130]) was of invaluable help: Proofs are cached in binary format so that it is possible to track which changes invalidate which proofs, or vice-versa, which proofs are unaffected by a particular change. Moreover, revalidating proofs is done on-demand, which admits refactoring specifications without the immediate need to reconsider large parts of the verification. KIV supports to replay previous proof attempts in order to reuse parts of the verification after a change or even in a different part of the development. In the course of this project, the procedure has been automated further, for instance checking and revalidating a whole subcomponent is just a single click now. Nevertheless, improved tool support for incremental refactoring of models is something we could benefit from in the future.

We note in this context that proof automation is *not* the limiting factor. The provers that exist today are certainly capable of dealing with the verification of large scale software systems. In our opinion, providing the right abstraction capabilities and streamlining the development workflow are much more important aspects.

First-Class Support for Modeling Concepts

The refinement theory and approach to power cuts developed in this thesis provides explicit modeling constructs for machines, submachines, operations with preconditions, crash specifications and recovery. As already discussed, these concepts could have been encoded in some form or the other (Section 4.6.3, Section 11.7.1). However, we are convinced that such encodings obscure the intentions (cf. the observation about control state ASMs in [28, Section 3.1]), and impede maintenance of the models.

Furthermore, proofs that can be carried out at the semantic level such as the compositionality in Section 5.3 or the reduction in Section 11.6 are harder or not possible with such encodings.

With respect to tool support, we have added data type like machines including their crash predicates and recovery as a specification mechanism to the KIV theorem prover. As a consequence, the proof obligations for forward simulation (Theorem 5.4), crash recovery (Lemma 11.16), and crash neutrality (Theorem 11.22) are generated automatically and adjusted whenever the models change. This lead to a significant boost in productivity.

Chapter 14

Conclusions and Outlook

Conclusions

Flashix is the first file system for flash memory that is proven functionally correct and both power cut safe. This thesis contributes a large part to its development, both on the theoretical side as well as the practical side.

The success of the project is based on 1) the incremental and modular approach that seamlessly integrates the verification of functional correctness as well as crash tolerance and 2) finding the right levels of abstraction at which file system concepts are modeled.

Development of verified software is an exercise in *design*, much more so than development of unverified software. It depends on the right choice of abstractions, modules and data structures whether verification is feasible at all. Complexity arises most in the *interaction* between different parts of the system. Local changes tend to have global impact.

Outlook

Currently, Flashix is not en par with the performance of existing file systems. The reason is not that its internal strategies and data structures are inefficient. Instead, it lacks two features that let the file system *appear* to be much faster:

Write-back caching is a technique to defer writing data to flash memory. Instead, writes are cached in main memory of time and flushed to the storage medium asynchronously (in the background). As a consequence, the latency of operations as it is visible to the user of a file system shrinks dramatically. From the perspective of formal verification, write-back caching is problematic when power cuts are considered. Even specifying what the correct behavior of a file system should be is unclear, initial approaches in e.g. [32] are based on rewriting the history of the whole system.

Concurrency of internal operations is a prerequisite to offload work into background operations. Extending the verification to a concurrent setting is a significant undertaking, in particular when the top-level operations should be thread-safe. At the same time, there are research opportunities to make concurrency verification scale to large scale systems. We have already taken care to make the semantics in this thesis compatible with the temporal logic RGITL [145] supported by KIV.

A major concern for the integration of new features such as write-back caches and concurrency is how much of the existing verification can be reused. The focus therefore shifts from the incremental development of one refinement hierarchy towards extending a given development.

Bibliography

[1] Intel flash file system core reference guide, version 1. Technical Report 304436001, Intel Corporation, 2004.

[2] Open NAND Flash Interface Specification. Intel Corporation et al., 2013. URL http://www.onfi.org/specifications.

[3] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. The IEEE and The Open Group, 2013. URL http://pubs.opengroup.org/onlinepubs/9699919799/.

[4] OMG Unified Modeling Language™, Version 2.5. The Object Management Group (OMG), 2015. URL http://www.omg.org/spec/UML/.

[5] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] J.-R. Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[7] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[8] S. Amani and T. Murray. Specifying a realistic file system. In *Proc. of the Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, 2015.

[9] 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. Heiserer. COGENT: Verifying high-assurance file system implementations. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188. ACM, 2016.

[10] K. Arkoudas, K. Zee, V. Kuncak, and M. C. Rinard. Verifying a file system implementation. In *Proc. of the International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 373–390. Springer, 2004.

[11] M. Nicolosi Asmundo. *Consistent composition of Abstract State Machine models*. PhD thesis, University of Catania, Italy, 2002.

[12] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the common algebraic specification language. *Theoretical Computer Science (TCS)*, 286(2):153–196, 2002.

[13] R.-J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

[14] R.-J. Back and J. von Wright. Trace refinement of action systems. In *Proc. of Concurrency Theory (CONCUR)*, volume 836 of *LNCS*, pages 367–384. Springer, 1994.

[15] J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the Tokeneer enclave protection software. In *Proc. of the International Symposium on Secure Software Engineering (ESSoS)*. IEEE, 2006.

[16] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Formal verification of a microkernel used in dependable software systems. In *Proc. of Computer Safety, Reliability, and Security (SAFECOMP)*, volume 5775 of *LNCS*, pages 187–200. Springer, 2009.

[17] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons learned from microkernel verification–specification is the new bottleneck. In *Proc. of Software and Systems Modeling*

(SSV), volume 102 of *EPTCS*, pages 18–32. Elsevier, 2012.

[18] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[19] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *Proc. of Formal Methods (FM)*, volume 1708 of *LNCS*, pages 369–387. Springer, 1999.

[20] Y. Bertot and P. Castéran. *Coq'Art: Interactive theorem proving and program development.* Springer, 2004.

[21] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98. ACM, 2016.

[22] S. Brookes. Full abstraction for a shared variable parallel language. In *Proc. of Logic in Computer Science (LICS)*, pages 98–109. IEEE, 1993.

[23] R. Bubel, C. C. Din, R. Hähnle, and K. Nakata. A Dynamic Logic with traces and coinduction. In *Proc. of Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 9323 of *LNCS*, pages 307–322. Springer, 2015.

[24] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, (7):23–50, 1972.

[25] A. Butterfield and A. Ó. Catháin. Concurrent models of flash memory device behaviour. In *Proc. of the Brazilian Symposium on Formal Methods (SBMF)*, volume 5902 of *LNCS*, pages 70–83. Springer, 2009.

[26] A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming (SCP)*, 74(4):219–237, 2009.

[27] E. Börger. The ASM refinement method. *Formal Aspects of Computing (FAC)*, 15(2):237–257, 2003.

[28] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In *Proc. of Computer Science Logic (CSL)*, volume 1862 of *LNCS*, pages 41–60. Springer, 2000.

[29] E. Börger and R. F. Stärk. *Abstract State Machines—A method for high-level system design and analysis.* Springer, 2003.

[30] E. Börger, A. Cisternino, and V. Gervasi. Ambient Abstract State Machines with applications. *Journal of Computer and System Sciences (JCSS)*, 78(3):939–959, 2012.

[31] A. Cau and B. Moszkowski. *ITL—Interval Temporal Logic.* Software Technology Research Laboratory, De Montfort University, Leicester, England, 2015. URL http://www.antonio-cau.co.uk/ITL/.

[32] H. Chen. *Certifying a crash-safe file system.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.

[33] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX Association, 2015.

[34] 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 the Symposium on Operating Systems Principles (SOSP)*. ACM, 2015.

[35] S. Cheng, J. Woodcock, and D. D'Souza. Using formal reasoning on a model of tasks for FreeRTOS. *Formal Aspecpts of Computing (FAC)*, 27(1):167–192, 2014.

[36] 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 Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[37] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2014. URL http://pdos.csail.mit.edu/6.828/2014/xv6.html.

[38] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in modular specifications for concurrent modules. In *Proc. of Mathematical Foundations of Programming Semantics (MFPS)*, pages 3–18. Elsevier, 2015. Invited Tutorial Paper.

[39] K. Damchoom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *Proc. of the Brazilian Symposium on Formal Methods (SBMF)*, volume 5902 of *LNCS*, pages 134–152. Springer, 2009.

[40] 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 the International Conference on Formal Engineering Methods (ICFEM)*, volume 5256 of *LNCS*, pages 25–44. Springer, 2008.

[41] W.-P. de Roever and K. Engelhardt. *Data refinement: model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[42] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[43] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific, 1995.

[44] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[45] T. Dinsdale-Young, L. Birkedaland P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *Proc. of Principles of Programming Languages (POPL)*, pages 287–300. ACM, 2013.

[46] S. Divakaran, D. D'Souza, A. Kushwah, P. Sampath, N. Sridhar, and J. Woodcock. Refinement-based verification of the FreeRTOS scheduler in VCC. In *Proc. of the International Conference on Formal Engineering Methods (ICFEM)*, volume 9407 of *LNCS*, pages 170–186. Springer, 2015.

[47] S. Divakaran, D. D'Souza, P. Sampath, N. Sridhar, and J. Woodcock. A theory of refinement for ADTs with functional interfaces. Technical Report 2015-04, Indian Institute of Science, Bangalore, India, 2015.

[48] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proc. of the Internet Measurement Conference (IMC)*, pages 475–488. ACM, 2014.

[49] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, A. Schierl, and W. Reif. Flashix web presentation and models. Institute of Software and Systems Engineering, Augsburg University, Germany, 2009–2016. URL `http://isse.de/flashix`.

[50] 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. Elsevier, 2012.

[51] 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.

[52] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular refinement for submachines of ASMs. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 188–203. Springer, 2014.

[53] 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.

[54] 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.

[55] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming (SCP)*, 2016. In Print.

[56] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–104, 2007.

[57] M. A. Ferreira, S. S. Silva, and J. N. Oliveira. Verifying Intel flash file system core specification. In *Proc. of the VDM/Overture Workshop*, pages 54–71, England, 2008. University of Newcastle

upon Tyne.

[58] I. Filipović, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science (TCS)*, 411(51-52):4379 – 4398, 2010.

[59] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[60] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: A roadmap. In *Proc. of the International Conference on Engineering Complex Computer Systems (ICECCS)*, pages 153–162. IEEE, 2008.

[61] 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.

[62] J. C. Freytag, F. Cristian, and B. Kähler. Masking System Crashes in Database Application Programs. In *Proc. of Very Large Data Bases (VLDB)*, pages 407–416. ACM, 1987.

[63] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. I. Siminiceanu. Model-Checking the Linux Virtual File System. In *Proc. of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 74–88. Springer, 2009.

[64] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing (FAC)*, 5(4):367–382, 1993.

[65] P. Gardner, G. Ntzik, and A. Wright. Local reasoning for the POSIX file system. In *Proc. of the European Symposium on Programming (ESOP)*, volume 8410 of *LNCS*, pages 169–188. Springer, 2014.

[66] A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 14 (12):1949–1983, 2008.

[67] G. Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(2), 1934.

[68] G. Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39 (3), 1935.

[69] T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI—Unsorted Block Images, 2006. URL http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf.

[70] S. Glesner. A proof calculus for natural semantics based on greatest fixed point semantics. In *Proc. of Compiler Optimization meets Compiler Verification (COCV)*, volume 132 of *EPTCS*, pages 73–93. Elsevier, 2005.

[71] A. Groce, G. Holzmann, R. Joshi, and R-G. Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Proc. of Constraints in Formal Verification (CFV)*, 2008.

[72] Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[73] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[74] D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, 2000.

[75] J. Hatcliff, G. T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, 2012.

[76] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 1–17. ACM, 2015.

[77] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European Symposium on Programming (ESOP)*, pages 187–196. Springer, 1986.

[78] M. Heisel. Specification of the UNIX file system: A comparative case study. In *Proc. of Algebraic Methodology and Software Technology (AMAST)*, volume 936 of *LNCS*, pages 475–

488. Springer, 1995.

[79] M. Heisel, W. Reif, and W. Stephan. A dynamic logic for program verification. In *Proc. of Logical Foundations of Computer Science (LFCS)*, volume 363 of *LNCS*, pages 134–145. Springer, 1989.

[80] 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.

[81] W. H. Hesselink and M. I. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing (FAC)*, 24(1):27–44, 2012.

[82] C. A. R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[83] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[84] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[85] C. A. R Hoare. Compensable Transactions. volume 9 of *NATO Security through Science Series - D: Information and Communication Security*, pages 116–134. IOS Press, 2007.

[86] C. A. R Hoare et al. *Communicating Sequential Processes*. Prentice Hall, 1985.

[87] M. Huisman and J. Jacobs. Java program verification via a hoare logic with abrupt termination. In *Proc. of Fundamental Approaches to Software Engineering (FASE)*, pages 284–303. Springer, 2000.

[88] A. Hunter. A brief introduction to the design of UBIFS, 2008. URL http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

[89] B. Jacobs and J. Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin—European Association for Theoretical Computer Science (EATCS)*, 62:222–259, 1997.

[90] C. Jones and J. Woodcock. Editorial to the mondex special Issue. *Formal Aspecpts of Computing (FAC)*, 20(1):1–3, 2008.

[91] R. Joshi and G. J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing (FAC)*, 19(2):269–272, 2007.

[92] E. Kang and D. Jackson. Formal modelling and analysis of a flash filesystem in Alloy. In *Proc. of Abstract State Machines, B, and Z (ABZ)*, volume 5238 of *LNCS*, pages 294–308. Springer, 2008.

[93] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File systems deserve verification too! *ACM Operating Systems Review*, 48(1):58–64, 2014.

[94] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[95] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *Transactions on Computer Systems (TOCS)*, 32(1):1–70, 2014.

[96] E. Koskinen and J. Yang. Reducing crash recoverability to reachability. In *Proc. of Principles of Programming Languages (POPL)*, pages 97–108. ACM, 2016.

[97] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[98] M. I. Lali. File system formalization: revisited. *International Journal of Advanced Computer Science*, 3(12):602–606, 2013.

[99] L. Lamport. *Specifying systems: the TLA$^+$ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

[100] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009.

[101] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[102] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A Study of Linux file system evolution. *ACM Transactions on Storage (TOS)*, 10(1):3:1–3:32, 2014.

[103] N. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[104] O. Marić and C. Sprenger. Verification of a transactional memory manager under hardware failures and restarts. In *Proc. of Formal Methods (FM)*, volume 8442 of *LNCS*, pages 449–464. Springer, 2014.

[105] J. Mauro and R. McDougall. *Solaris internals: core kernel components*. Prentice Hall, 2001.

[106] A. A. McEwan and J. Woodcock. Unifying theories of interrupts. In *Proc. of Unifying Theories of Programming (UTP)*, volume 5713 of *LNCS*, pages 122–141. Springer, 2010.

[107] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[108] C. Morgan and B. Sufrin. Specification of the UNIX filing system. *Transactions on Software Engineering*, 2:128–142, 1984.

[109] W. Mostowski. A case study in formal verification using multiple explicit heaps. In *Proc. of Formal Techniques for Distributed Systems (FORTE)*, volume 7892 of *LNCS*, pages 20–34. Springer, 2013.

[110] B. Moszkowski. *Reasoning about digital circuits*. PhD thesis, Stanford University, CA, United States, 1983.

[111] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, 2014.

[112] J. T. Mühlberg and G. Lüttgen. Verifying compiled file system code. *Formal Aspecpts of Computing (FAC)*, 24(3):375–391, 2012.

[113] M. Najafzadeh. *The analysis and co-design of weakly-consistent applications*. PhD thesis, Université Pierre et Marie Curie, France, 2016.

[114] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The CISE tool: proving weakly-consistent applications correct. In *Proc. of Principles and Practice of Consistency for Distributed Data (PaPoC)*. ACM, 2016.

[115] K. Nakata and T. Uustalu. Trace-Based coinductive operational semantics for While. In *Proc. of Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 375–390. Springer, 2009.

[116] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. *ACM SIGPLAN Notices*, 45(1):261–274, 2010.

[117] M. Nicolosi Asmundo and E. Riccobene. Consistent integration for sequential Abstract State Machines. In *Proc. of Abstract State Machines, Advances in Theory and Practice (ASM)*, volume 2589 of *LNCS*, pages 324–340. Springer, 2003.

[118] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science (TCS)*, 13(1):85–108, 1981.

[119] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[120] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Proc. of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 9458 of *LNCS*, pages 169–188. Springer, 2015.

[121] L. O'Connor-Davis, G. Keller, S. Amani, T. Murray, G. Klein, Z. Chen, and C. Rizkallah. CDSL version 1: Simplifying verification with linear types. Technical report, NICTA, Sydney, Australia, 2014.

[122] J. N. Oliveira and M. A. Ferreira. Alloy meets the algebra of programming: A case study. *IEEE Transactions on Software Engineering*, 39(3):305–326, 2013.

[123] 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 Hardware and Software: Verification and Testing (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.

[124] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Crash-safe refinement for a verified flash file system. Technical Report 2014-02, University of Augsburg, Germany, 2014.

[125] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash consistency. *Communications of the ACM*, 58(10):46–51, 2015.

[126] M.-L. Potet and Y. Rouzaud. Composition and refinement in the B-method. In *Proc. of the B Conference*, volume 1393 of *LNCS*, pages 46–65. Springer, 1998.

[127] V. R. Pratt. Semantical consideration on Floyd-Hoare logic. In *Proc. of the Symposium on Foundations of Computer Science (SFCS)*, pages 109–121. IEEE, 1976.

[128] V. R. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In *Proc. of Concurrency Theory (CONCUR)*, volume 197 of *LNCS*, pages 180–196. Springer, 1984.

[129] G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Proc. of the Aerospace Conference*, pages 4186–4199. IEEE, 2005.

[130] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction—A Basis for Applications*, volume II, pages 13–39. Kluwer, 1998.

[131] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[132] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2015.

[133] C. Rizkallah. *Verification of program computations*. PhD thesis, Saarland University, Germany, 2015.

[134] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[135] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[136] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[137] G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, University of Ulm, Germany, 1999. URL https://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/staff/schellhorn/publications. English translation, original title *Verifikation Abstrakter Zustandsmaschinen*.

[138] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001.

[139] G. Schellhorn. ASM Refinement and generalizations of forward simulation in data refinement: a comparison. *Journal of Theoretical Computer Science (TCS)*, 336(2–3):403–435, 2005.

[140] G. Schellhorn. ASM refinement preserving invariants. *Journal of Universal Computer Science (J.UCS)*, 14(12):1929–1948, 2008.

[141] G. Schellhorn. Completeness of fair ASM refinement. *Science of Computer Programming (SCP)*, 76(9):756–773, 2009.

[142] G. Schellhorn, K. Stenzel, D. Haneberg, W. Reif, B. Tofan, G. Ernst, and J. Pfähler. *A practical course on KIV*. Institute of Software and Systems Engineering, Augsburg University, Germany. URL http://isse.de/kiv/documentation.pdf.

[143] G. Schellhorn, J. Derrick, and H. Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Transactions on Computational Logic (TOCL)*, 15(4):31:1–31:37, 2014.

[144] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper.

[145] 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 (AMAI)*, 71:1–44, 2014.

[146] G. Schellhorn, G. Ernst, J. Pfähler, and W. Reif. A relational encoding for a clash-free subset of ASMs. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 9675 of *LNCS*, pages 237–243. Springer, 2016.

[147] 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.

[148] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proc. of File and Storage Technologies (FAST)*. USENIX Association, 2005.

[149] R. F. Stärk and S. Nanchen. A complete logic for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 7(11):981–1006, 2001.

[150] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[151] P. Taverne and C. Pronk. RAFFS: Model checking a Robust Abstract Flash File Store. In *Proc. of the International Conference on Formal Engineering Methods (ICFEM)*, volume 5885 of *LNCS*, pages 226–245. Springer, 2009.

[152] B. Tofan. *Compositional Concurrent Program Verification with RGITL*. PhD thesis, Augsburg University, Germany, 2014.

[153] H-W. Tseng, L. Grupp, and S. Swanson. Understanding the impact of power loss on flash memory. In *Proc. of the Design Automation Conference (DAC)*, pages 35–40. ACM, 2011.

[154] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. *ACM SIGPLAN Notices*, 42(1):97–108, 2007.

[155] S. Wang. Certifying checksum-based logging in the RapidFSCQ crash-safe filesystem. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.

[156] D. A. Wheeler. How to prevent the next Heartbleed, 2015. URL http://www.dwheeler.com/essays/heartbleed.html.

[157] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14 (4):221–227, 1971.

[158] J. Woodcock. The verified software repository. Marktoberdorf Summer School Lecture Notes, 2008.

[159] J. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall, 1996.

[160] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, 2009.

[161] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proc. of Operating Systems Design and Implementation (OSDI)*, pages 273–288. USENIX Association, 2004.

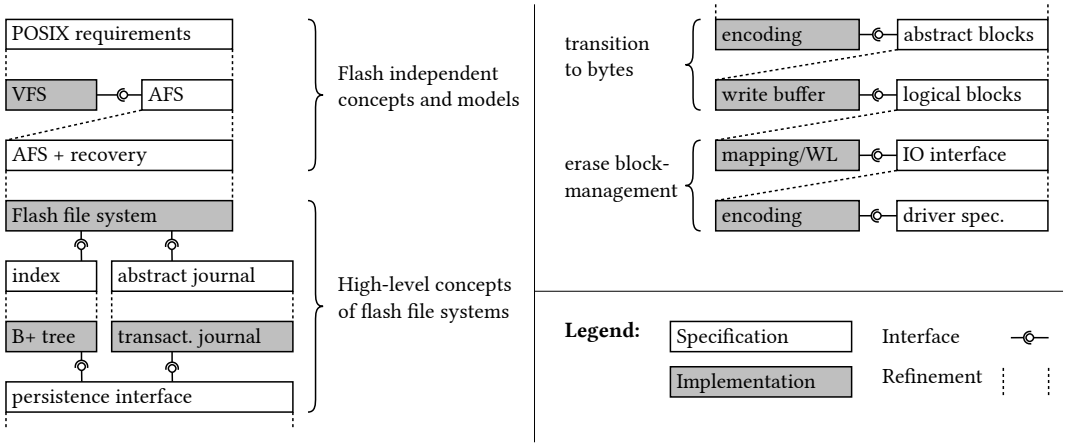
[162] S. Zenzaro. *On modularity in Abstract State Machines*. PhD thesis, Pisa University, Italy, 2016.

[163] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge. Understanding the robustness of SSDs under power fault. In *Proc. of File and Storage Technologies (FAST)*, pages 271–284. USENIX Association, 2013.

Appendix A

Model Summary

This appendix gives a summary of the formal models in terms of their interfaces, the concepts addressed, and their internal state.



POSIX

Chapter 7

State

t: `Tree`
fs: `Fid` $\rightarrow\!\!\!\!\!\!+$ `FileData`
oh: `Nat` $\rightarrow\!\!\!\!\!\!+$ `Handle`

Concepts

- paths, file handles
- files, directories
- hard links, orphans
- access permissions

Interface

`posix_create`($path$, md; err)
`posix_link`($from$, to; err)
`posix_unlink`($path$; err)
`posix_rename`($from$, to; err)
`posix_mkdir`($path$, md; err)
`posix_rmdir`($path$; err)
`posix_readmeta`($path$; md, err)
`posix_writemeta`($path$, md; err)
`posix_readdir`($path$; $names$, err)
`posix_open`($path$, $mode$; fd, err)
`posix_close`(fd; err)
`posix_read`(fd; buf, len, err)
`posix_write`(fd, buf; len, err)
`posix_truncate`($path$; len, err)

Virtual File System (VFS)

Chapter 8

State

oh: `Nat` $\rightarrow\!\!\!\!\!+$ `Handle`

Concepts

- generic vs. flash specific aspects
- stepwise path lookup
- pagewise read/write

Interface

see POSIX

Internal

`vfs_walk`($path$; ino, err)
`vfs_putino`(ino)
`vfs_may_`$*$(ino; $inode$, $dent$, $\cdots$, err)

Abstract File System (AFS)

Chapter 8

State

$dirs$: `Ino` $\rightarrow\!\!\!\!\!+$ `Dir`
$files$: `Ino` $\rightarrow\!\!\!\!\!+$ `File`

Concepts

- files, directories, pages
- hard links, orphans
- bookkeeping of sizes and counters

Interface

`afs_lookup`(ino; $dent$, err)
`afs_iget`(ino; $inode$, err)
`afs_create`(md; $inode$, $dent$, err)
`afs_link`($dent$; $pinode$, $cinode$, $dent'$, err)
`afs_unlink`(; $inode$, $dent$, err)
`afs_rename`(; $pinode$; $cinode$, $dent$,
$pinode'$, $cinode'$, $dent'$, err)
`afs_mkdir`(md; $inode$, $dent$, err)
`afs_rmdir`(; $inode$, $dent$, err)
`afs_write_inode`(; $inode$, err)
`afs_readdir`($inode$; $names$, err)
`afs_readpage`($inode$; $page$, err)
`afs_writepage`($inode$, $page$; err)
`afs_truncate`($inode$; len, err)
`afs_evict`(ino)

Flash File System Core (FFS)

Chapter 9

State

ro: $\mathit{Set}\langle \mathit{Key} \rangle$

Concepts

- out-of-place updates
- garbage collection
- logging, crashes, recovery

Interface

see AFS

Internal

$\texttt{fs\_commit}(; err)$
$\texttt{fs\_recover}(\,)$
$\texttt{fs\_garbage\_collect}(\,)$

Journal Specification

Chapter 9

State

fs: $\mathit{Address} \nrightarrow \mathit{Node}$
log: $\mathit{List}\langle \mathit{Address} \rangle$
fo: $\mathit{Set}\langle \mathit{Key} \rangle$

Interface

$\texttt{jnl\_get}(adr;\ nd, err)$
$\texttt{jnl\_append}_n(nd_1, .., nd_n;\ adr_1, .., adr_n,\ err)$

Index Specification

Chapter 9

State

ri, fi: $\mathit{Key} \nrightarrow \mathit{Address}$

Interface

$\texttt{idx\_lookup}(key; adr, exists)$
$\texttt{idx\_store}(key, adr)$
$\texttt{idx\_remove}(key)$
$\texttt{idx\_dentries}(key; keys)$
$\texttt{idx\_trunc}(key, n)$
$\texttt{idx\_newino}(; key)$