Presented at EuroSys Doctoral Workshop 2012, Bern, Switzerland.

Towards a Fully Verified File System

Sidney Amani
NICTA & University of New South Wales
sidney.amani@nicta.com.au

Leonid Ryzhyk Toby Murray
NICTA & University of New South Wales
leonid.ryzhyk@nicta.com.au
toby.murray@nicta.com.au

1. Motivation

Implementation defects in file systems can lead to disastrous data-loss. This situation has drawn the attention of researchers for a long time, yet file system bugs are still very common [4, 6, 8]. Even well established file systems like Ext2 and Ext3, which have not been extended with new features for years, occasionally get patched to fix implementation flaws[3].

Previous work on file system verification was mostly based on model checking[4, 6, 8] which aims at preventing certain classes of bugs such as buffer overflows or NULL pointer dereferences. File systems are central part of operating systems, expectations from such critical components go beyond the mere absence of generic programming errors.

The usual approach to obtain functional correctness guarantees is to prove the implementation correct using an interactive theorem prover. This approach enables strong functional correctness guarantees encompassing most properties checked by model checking tools. However, as opposed to model checking, formal verification using a theorem prover cannot be fully automated and the required interactions with the user make the approach slow and tedious.

Previous manual formal verification attempts suffered from the overwhelming size and complexity of file system implementations [1, 2, 5, 7]. In order to prove difficult properties about the file system, this research had to introduce serious limitations resulting in oversimplified and unusable file systems.

In this paper, we propose a solution to deal with the size and the complexity of a realistic file system implementation in order to prove its functional correctness. As opposed to previous research, we only consider simplifications if the resulting file system remains usable and reasonably efficient. Moreover the functional correctness proof only makes sense if it can be related to the C implementation. To this end, we will automatically generate a C file system implementation from our provably correct specification.

2. File system verification: challenges

The ultimate goal of the project is to obtain a full functional correctness proof of a file system implementation. One way to define the correct behaviour of a system is to write an abstract specification describing the behaviours expected from it. Then proving that the system is correct is proving a refinement between the implementation and the specification, i.e. that every behaviour exhibited by the implementation has an equivalent behaviour in the abstract specification. Such equivalence is proved correct mathematically using an interactive theorem prover.

Previous research [1, 2, 5, 7] has proved refinement between two or more high-level file system specifications but none of these efforts have managed to relate the specification to an implementation of a realistic file system. File system behaviours can easily be described with high level data structure such as sets, lists and maps. This representation enables proving the correctness of the file system design while abstracting away implementation details. However, the real complexity arises when introducing implementation details that specify a concrete layout for these data structures. Many properties that are difficult to verify emerge when adding these implementation details. For instance, describing a directory as a set of of directory entries makes it trivial to verify that all entries in a directory are unique. Specifying how these entries are stored, how data structures are packed in disk blocks and proving that the implementation never adds twice the same entry is much harder.

Another important issue is error handling. Previous research completely ignored errors such as allocation failures (memory or disk block). A real file system implementation has to deal with such failures and recent research has shown[4] that they play an important role in file system complexity.

A realistic file system specification must reflect all these implementation details. Previous research has kept these implementation details away as a refinement proof for a specification that carries all these notions would require too much effort and appears infeasible.

3. Towards a verified file system

We propose a practical methodology for complete functional verification of a realistic file system. To overcome the complexity that arises when we prove the refinement between

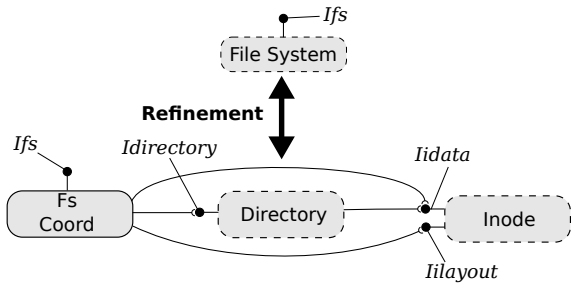

Figure 1. Example of file system decomposition

two specifications, we propose to decompose a specification by functionality.

The decomposition process proceeds as follow:

- Split a specification into multiple components.
- Specify well-defined interfaces between them.
- Specify the behaviour of each component in the decomposition. At this stage we can prove the refinement between the combination of components and the monolithic specification.
- Refine each component individually, by possibly repeating the decomposition process for each of them.

Beside splitting a complex problem into multiple smaller ones, decomposing a specification also facilitates teamwork as different verification engineers can work on independent proofs. Another advantage is that multiple file systems that expose common behaviours, can reuse the same components and therefore the same proof.

On the other hand, decomposing a specification influences the structure of the implementation as a component communicates exclusively through well-defined interfaces. Simple interfaces confines the verification effort but may restrict the implementation components and impact their performance. Another drawback is that modifying an interface can potentially affect multiple proofs, a simple change at this level may introduce several proof updates. Lastly, some behaviours cannot be easily decomposed, concurrency falls in this category.

3.1 Example of decomposition

Figure 1 shows a file system specification decomposed into 3 components. Dashed borders indicate that the component is going to be decomposed in a further refinement, whereas solid borders denote components that contain enough implementation details to be related to the file system implementation. Components interact solely through interfaces (e.g. *Ifs*).

The file system coordinator (`FS Coord`) implements all file system operations by delegating the work to either `Directory` or `Inode`. `Directory` manages all directory operations such as add, remove and list directory entries in directories.

`Inode` implements operations to read/write data allocated to an inode (*Iidata*) and operations to create/delete inodes (*Iilayout*).

By repeating the decomposition process for `Directory` and `Inode`, we are able to split the file system functionality into small components that require simple proof obligations and their correctness can be proved individually. When we decompose a specification, implementation details are omitted, they only become relevant when proving refinement between the component implementation and its specification.

3.2 Expected research contributions

- First functional correctness proof of a realistic file system implementation.
- An approach to file system verification by decomposition.

References

[1] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *In Sixth International Conference on Formal Engineering Methods (ICFEM04), volume 3308 of LNCS*, pages 8–12, 2004.

[2] Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in event-b. In *SBMF 2009*, volume 5902, pages 134–152, 2009. Springer LNCS 5902.

[3] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 2012.

[4] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 14:1–14:16, Berkeley, CA, USA, 2008.

[5] Wim H. Hesselink and M. I. Lali. Formalizing a hierarchical file system. *Electron. Notes Theor. Comput. Sci.*, 259:67–85, December 2009.

[6] Cindy Rubio-González and Ben Liblit. Defective error/pointer interactions in the linux kernel. In Frank Tip, editor, *International Symposium on Software Testing and Analysis*, Toronto, Ontario, Canada.

[7] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the ubifs file system for flash memory. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 190–206, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 10–10, Berkeley, CA, USA, 2006. USENIX Association.