A Mini Challenge: Build a Verifiable Filesystem

Rajeev Joshi and Gerard J. Holzmann

Laboratory for Reliable Software**,
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA 91109, USA

{Rajeev.Joshi,Gerard.Holzmann}@jpl.nasa.gov
http://eis.jpl.nasa.gov/lars

Abstract. We propose tackling a "mini challenge" problem: a nontrivial verification effort that can be completed in 2-3 years, and will help establish notational standards, common formats, and libraries of benchmarks that will be essential in order for the verification community to collaborate on meeting Hoare's 15-year verification grand challenge. We believe that a suitable candidate for such a mini challenge is the development of a filesystem that is *verifiably* reliable and secure. The paper argues why we believe a filesystem is the right candidate for a mini challenge and describes a project in which we are building a small embedded filesystem for use with flash memory.

A mini challenge

The verification grand challenge proposed by Hoare [1] sets the stage for the program verification community to embark upon a collaborative effort to build verifiable programs. At a recent workshop in Menlo Park [2], there seemed to be a consensus that a necessary stepping stone to such an effort would be the development of repositories for sharing specifications, models, implementations, and benchmarks so that different tools could be combined and compared.

We believe that the best way of reaching agreement on common formats and forging the necessary collaborations to build such a repository is to embark upon a shorter-term "mini challenge": a nontrivial verification project that can nonetheless be completed in a short time. An ideal candidate for such a mini challenge would have several characteristics: (a) it would be of sufficient complexity that traditional methods such as testing and code reviews are inadequate to establish its correctness, (b) it would be of sufficient simplicity that specification, design and verification could be completed by a dedicated team in a relatively short time, say 2-3 years, and (c) it would be of sufficient importance

** The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

that successful completion of the mini challenge would have an impact beyond the verification community.

At the Menlo Park workshop, some participants (notably Amir Pnueli) suggested that a suitable candidate would be the verification of the kernel[1] of the Linux operating system [3]. While the task of verifying the Linux kernel undoubtedly meets conditions (a) and (c) above, it does not meet condition (b). In fact, given that the current Linux kernel is well over 4 million lines of source code, it seems a tall order to write a formal specification for it within 2 years, much less verify the correctness of the implementation. Instead, we propose that a more suitable candidate for such a mini challenge would be the development of a verifiable filesystem. We believe there are several reasons why a filesystem is more attractive as a first target for verification than an operating system kernel.

Firstly, most modern filesystems have a clean, well-defined interface, conforming to the POSIX standard [4], which has been in use for many years. Thus writing a formal specification for a POSIX-compliant filesystem would require far less effort than writing a kernel specification. In fact, one could even write an abstract reference filesystem implementation which could be used as the specification for a verification proof based on refinement.

Secondly, since the underlying data structures and algorithms used in filesystem design are very well understood, a verifiable filesystem implementation could conceivably be written from scratch. Alternatively, researchers could choose any of several existing open-source filesystems and attempt to verify them. This makes filesystem verification attractive, since it allows participation by both those researchers interested in *a posteriori* verification, as well as those interested in “constructing a program and its proof hand-in-hand”.

Thirdly, although filesystems comprise only a small portion of an operating system, they are complex enough that ensuring reliability in the presence of concurrent accesses and unexpected power failures is nontrivial. Indeed, recent work by Yang et al shows that many popular filesystems in widespread use have serious bugs that can have devastating consequences, such as deletion of the system root directory [7].

Finally, since almost all data on modern computers is now managed by filesystems, their correctness is of great importance, both from the standpoint of reliability as well as security. Development of a verified filesystem would therefore be of great value even beyond the verification community.

Directions and Challenges

The goal of the proposed mini challenge is to build a *verifiable* filesystem. In particular, we are interested in the problem of how to write a filesystem whose correctness can be checked using automated verification tools. After decades of experience with automatic program verification, we know that such an effort inevitably requires that key design knowledge be captured and expressed in machine readable forms that can be used to guide the verification tools. This

[1] Actually, Pnueli suggested verifying “Linux”; we assume he meant the Linux kernel.

includes (a) a formal behavioral specification of the functionality provided by the filesystem, (b) a formal elaboration of the assumptions made of the underlying hardware, and (c) a set of invariants, assertions and properties concerning key data structures and algorithms in the implementation. We discuss each of these artifacts below.

Specification Most modern filesystems are written to comply with the POSIX standard [4] for filesystems. This standard specifies a set of function signatures (such as `creat, open, read, write`), along with a behavioral description of each function. However, these behavioral descriptions are given as informal English prose, and are therefore too ambiguous and incomplete to be useful in a verification effort. The first task therefore is to write a formal specification of the POSIX standard (or at least of a substantial portion of the standard) either as a set of logical properties or as an abstract reference implementation. Such formal specifications have been written in the past: for instance, by Morgan and Sufrin [5], who wrote a specification of the UNIX filesystem in Z, and by Bevier, Cohen and Turner [6], who wrote a specification for the Synergy filesystem in Z (and also partially in ACL2). Although these specifications did not completely model POSIX behavior (for instance, neither completely modeled error codes, nor file permissions), they could serve as starting points for developing a more complete specification.

Assumptions about underlying hardware In order to provide a rigorous formal statement of the properties of the filesystem (especially its robustness with respect to power failure), it is necessary to rely on certain behavioral assumptions about the underlying hardware. In order to make the filesystem useful, it is necessary to understand what assumptions can reasonably be made about typical hardware such as hard drives or flash memory. These assumptions need to be explicitly identified and clearly stated, as opposed to used implicitly in correctness proofs (as is often the case). In the ideal situation, the filesystem would be usable with different types of hardware, perhaps providing different reliability guarantees.

Properties of data structures and procedures As noted before, an attractive feature of the proposed mini challenge is that one could either write a verifiable filesystem from scratch, or verify an available filesystem. In either case, however, in order to use automatic checking tools to prove nontrivial correctness properties of the implementation, it will inevitably be necessary to identify and express design properties such as data structure invariants, annotations describing which locks protect which data, and pre- and post-conditions for library functions. Most typical filesystems require use of many common data structures such as hash tables, linked lists and search trees. A proof of filesystem correctness would therefore result in development of libraries of formally stated properties and proofs about these data structures, which would be useful in other verification efforts as well.

A reliable flash filesystem for flight software

At the NASA/JPL Laboratory for Reliable Software (LaRS), we are interested in the problem of building reliable software that is less reliant on following traditional ad-hoc processes and more reliant on use of automated verification tools. As part of this effort, we are currently engaged in a pilot project to help build a reliable filesystem for flash memory, for use as nonvolatile storage on board future missions.

Flash memory has recently become a popular choice for use on spacecraft as nonvolatile storage for engineering and data products, since it has no moving parts, consumes low power and is easily available. There are two common types of flash memory, NAND flash and NOR flash [8]. While NOR flash is more reliable and easier to program, it has lower density and poor write and erase times, and is therefore less attractive as a data storage device. While it is possible to design flight software to use flash memory directly as a raw device, it is typically much easier to write robust flight software on top of a filesystem layer that provides common file operations for creating, reading and writing files and directories. In fact, the flight software on several recent NASA missions, such as the Mars Exploration Rovers and Deep Impact, uses a filesystem to access flash memory.

Building a robust flash filesystem, however, is a nontrivial task. Performance dictates the use of caches and write buffers, which increase the danger of inconsistencies in the presence of concurrent thread accesses and unexpected power failures. To add to the challenge, flash memory, especially NAND flash memory, requires certain additional issues to be addressed such as arbitrary bit flips, blocks that unexpectedly become “bad” (i.e., permanently unusable), and limited lifetimes (block usually become bad after they have been erased a certain number of times, typically 100,000). In addition, a flash filesystem written for use on a spacecraft must obey additional constraints; for instance, flight software is typically allowed to allocate memory only during initialization.

The goal of our pilot project is to build a robust flash filesystem by following a design methodology that is based on documenting as much as possible in a machine readable form that is amenable to automatic verification. Thus the intent is not only to build a working filesystem, but also to produce key design documents in machine-readable forms that can be used by automated verification tools. Although less ambitious than the mini challenge we have described above (which is aimed at building a general purpose filesystem), our project has similar interests and goals with the mini challenge we have proposed.

Summary

An important first step toward the Verification Grand Challenge is the development of a repository containing specifications, models and implementations. We believe the best way to develop this repository is to tackle a “mini challenge” that can be completed in a short period of time, around 2-3 years. An excellent candidate for such a mini challenge seems to be the development of a verifiable filesystem that is both reliable and secure. Since filesystems are well-defined and well-understood, different research teams can take different approaches to building such a verifiable filesystem, from building it from scratch to verifying one of many available filesystems. We believe that the problem is well-suited as a mini challenge for the verification community and will serve as a starting point for the grand verification challenge.

References

1. Tony Hoare, *The Verifying Compiler: A Grand Challenge for Computing Research*, Journal of the ACM, 50(1), January 2003, pp. 63–69.
2. *Workshop on the Verification Grand Challenge*, SRI International, Menlo Park, CA, February, 2005. See `http://www.csl.sri.com/users/shankar/VGC05` .
3. Amir Pnueli, *Looking Ahead*, Presentation at the Workshop on The Verification Grand Challenge, SRI International, Menlo Park, CA, February, 2005. Slides available at `http://www.csl.sri.com/users/shankar/VGC05/pnueli.pdf` .
4. The Open Group, *The POSIX 1003.1, 2003 Edition Specification*, available online at `http://www.opengroup.org/certification/idx/posix.html` .
5. Carroll Morgan and Bernard Sufrin, *Specification of the UNIX Filing System*, IEEE Transactions on Software Engineering, Vol SE-10, No.2, March 1984, pp. 128–142.
6. William R. Bevier, Richard Cohen, and Jeff Turner, *A Specification for the Synergy File System*, Technical Report 120, Computational Logic, Inc., September 1995.
7. J. Yang, P. Twohey, D. Engler, and M. Musuvathi, *Using Model Checking to Find Serious File System Errors*, Proceedings of the Conference on Operating Systems Design and Implementation (OSDI), San Francisco, December 2004, pp. 273–288.
8. Data I/O, *A Collection of NAND Flash Application Notes, Whitepapers and Articles*, available at `http://www.data-io.com/NAND/NANDApplicationNotes.asp` .