DOI 10.1007/s00165-006-0022-3
BCS © 2007
Formal Aspects of Computing (2007)

Formal Aspects of Computing

A mini challenge: build a verifiable filesystem*

Rajeev Joshi[1,2], Gerard J. Holzmann[1]

[1]Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, USA.
E-mail: Rajeev.Joshi@jpl.nasa.gov; Gerard.J.Holzmann@jpl.nasa.gov
[2]4800 Oak Grove Drive, MS 301-285, Pasadena, CA 91109, USA

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.

Keywords: Verification grand challenge, Filesystem design, Formal verification

1. A mini challenge

The verification grand challenge proposed by Hoare [Hoa03] sets the stage for the program verification community to embark upon a collaborative effort to build verifiable programs. At recent meetings in Menlo Park [VGC05] and in Zurich [VSTTE], 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 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 [Pnu05]. 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 four 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

Correspondence and offprint requests to: Rajeev Joshi, E-mail: Rajeev.Joshi@jpl.nasa.gov, URL:http://eis.jpl.nasa.gov/lars

* 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.

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

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 [POSIX], 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, unexpected power losses and hardware failure is nontrivial. Indeed, recent work by Yang et al. [YTEM04] shows that many popular filesystems in widespread use have serious bugs that can have devastating consequences, such as deletion of the system root directory.

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.

2. Directions and challenges

The goal of our 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 proving nontrivial correctness properties of a modern filesystem inevitably requires that key design knowledge be captured and expressed in machine readable forms in order to guide verification tools. This 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.

2.1. Specification

Most modern filesystems are written to comply with the POSIX standard [POSIX] 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 [MoS84], who wrote a specification of the UNIX filesystem in Z, and by Bevier et al. [Bev95], 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.

2.2. Assumptions about 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.

2.3. 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.

3. A reliable flash filesystem for embedded systems

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 develop 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 [Dataio]. 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. Thus NOR flash is typically used for storing data that does not change too often, such as executable binaries and configuration parameters. In contrast, NAND flash has higher density and better write and erase times, and is therefore a more attractive medium for storing data that changes more often, such as telemetry and data collected by on-board instruments.

While it is possible to use flash memory directly as a raw device, it is typically much easier to write robust flight software if the flash is accessed through a filesystem interface that provides operations for creating, reading, and writing files and directories. In fact, several recent NASA missions, such as Mars Pathfinder, the twin Mars Exploration Rovers and the Deep Impact spacecraft, have used POSIX-compliant filesystems 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, has certain failure modes that need to be addressed in software, such as arbitrary bit flips, blocks that unexpectedly become "bad" (i.e., permanently unwritable), and limited block lifetimes (blocks are likely to become bad after they have been erased a certain number of times, typically 10^5 – 10^6 times). In addition, spacecraft flight software must be written according to certain rigid constraints; for instance, it may allocate memory from the heap only during initialization, and should use a (statically) bounded amount of stack space.

Perhaps it is not surprising, then, that recent experience at JPL with flash filesystems has shown that most available filesystems are not reliable enough for use in critical applications like flight software. In large part, this is the primary motivation for our interest in building a reliable filesystem. 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.

4. 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 use any of a wide range of 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

[Hoa03] Hoare T (2003) The verifying compiler: a grand challenge for computing research. J ACM 50(1):63–69

[VGC05] Workshop on the verification grand challenge (2005) SRI international, Menlo Park, CA. See `http://www.csl.sri.com/users/shankar/VGC05`

[VSTTE] Conference on Verified software: theories, tools, experiments. Eidgenössische Technische Hochschule Zürich, Zürich, 10–13, October 2006. See `http://vstte.ethz.ch`

[Pnu05] Pnueli A (2005) Looking Ahead. Presentation at the Workshop on The Verification Grand Challenge, SRI International, Menlo Park, CA. Slides available at `http://www.csl.sri.com/users/shankar/VGC05/pnueli.pdf`

[POSIX] The Open Group (2003) The POSIX 1003.1, 2003 edition specification. available online at `http://www.opengroup.org/certification/idx/posix.html`

[MoS84] Morgan C, Sufrin B (1984) Specification of the UNIX filing system. IEEE Transa Softw Engi SE-10(2):128–142

[Bev95] Bevier WR, Cohen R, Turner J (1995) A specification for the synergy file system. Technical Report 120, Computational Logic, Inc., September 1995

[YTEM04] Yang J, Twohey P, Engler D, Musuvathi M (2004) Using model checking to find serious file system errors. In: *Proceedings of the conference on operating systems design and implementation (OSDI)*, San Francisco, December 2004, pp 273–288

[Dataio] Data I/O A collection of NAND flash application notes, whitepapers and articles. available at `http://www.data-io.com/NAND/NANDApplicationNotes.asp`

Received : 13 October 2006

Revised : 6 November 2006

Accepted: 6 November 2006 by C. B. Jones and J. C. P. Woodcock