POSIX and the Verification Grand Challenge: a roadmap

Leo Freitas, Jim Woodcock
Department of Computer Science
University of York, YO10 5DD, UK

Andrew Butterfield
School of Computer Science and Statistics
Trinity College Dublin, Dublin 2, Ireland

E-mail: {leo, jim}@cs.york.ac.uk, Andrew.Butterfield@cs.tcd.ie

Abstract

We present a research roadmap for the second pilot project in the Verified Software Grand Challenge on formally verified POSIX file stores. The work is inspired by the requirements for NASA's forthcoming Mars Rover missions. The roadmap describes an integrated and comprehensive body of work, including current work, as well as further opportunities for collaboration.

1. Introduction

Recent advances in theory and tool support have inspired industrial and academic researchers to join up in an international Grand Challenge (GC) in Verified Software [22]. With tools backed by mature theory, formal methods are becoming more effective, and their use is easier to justify, not just as an academic or legal requirement, but as a business case. That is, despite the initial extra effort, the gains given by formal methods in terms of reliability, accountability, and precision, can save money. Also, as tool maturity rises, levels of expertise from the user tend to decrease, hence helping to de-skill the process and making it cheaper to industrialise.

Work has started with the creation of a Verified Software Repository (VSR) with two principal aims: (i) assembling a collection of verified software components; and (ii) performing industrial-scale verification experiments with theoretical significance and tool-support impact [11]. A research roadmap for the entire challenge is hosted at SRI's Computer Science Laboratory (see `qpq.csl.sri.com`).

Related work The first VSR pilot project experiment took place during 2006: the successful mechanisation of a sanitised version of the first ITSEC Level 6 high-integrity smart-card banking application: Mondex [49]. In that experiment, seven groups used different theories and tools to mechanise the specification of Mondex's security properties, the protocol that implements these properties, and to verify correctness of the protocol [52].

In the next section, we discuss the *POSIX* pilot project. The roadmap structure is given in Section 3. After that, we present the pillar of our work in Section 4. The *POSIX* standard is discussed in Section 5, where a brief overview of fault tolerance aspects via IBM CICS is given in Section 6. Section 7 decribes current efforts to standardise flash memory devices. Some conlcusions are drawn in Section 8.

2. What makes a pilot project?

In [34], Joshi & Holzmann suggest a pilot project for the GC. They characterise, motivate, and justify an interesting verification mini-challenge. It breaks the GC into smaller pilot projects, where each has the following features: (i) it would be of sufficient complexity that traditional methods, such as testing and code reviews, are inadequate to establish its correctness; (ii) 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 (iii) it would be of sufficient importance that successful completion would have an impact beyond the verification community, to both academia and industry.

At the Menlo Park workshop at SRI [48], the *POSIX* file-store interface of the Linux Kernel [33] was suggested as a candidate pilot project. The suggestion involved a small subset of *POSIX* suitable for flash-hardware with strict fault-tolerant requirements to be used by forthcoming *NASA* missions. Due to the nature of the environment this small subset would run in, two important robustness requirements for fault-tolerance were later agreed [12]: (i) no corruption in the presence of unexpected power loss; and (ii) recovery from faults specific to flash hardware (*i.e.*, bad blocks, bit corruption, wear-levelling, *etc*). In recovery from power loss in particular, they require the file system to be reset-reliable in the following sense: if an operation `Op` is in progress at the time of a power loss, then on reboot, the file system state will be as if `Op` either has successfully completed or has never started.

978-0-7695-3139-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICECCS.2008.35

***POSIX* file system** The choice of the *POSIX* file-system interface is interesting for various reasons: (i) it is a clean, well-defined, and standard interface that has been stable for many years; (ii) the data structures and algorithms required are well understood; (iii) although a small part of an operating system, it is complex enough in terms of reliability guarantees, such as unexpected power loss, concurrent access, or data corruption; and (iv) modern information technology is massively dependent on reliable and secure information availability. All these reasons go beyond the verification community interest, as well as the intended initial use on forthcoming NASA Mars Rover missions, as developed by the Jet Propulsion Laboratory (JPL).

An initial subset of *POSIX* file systems [41] has been chosen for the pilot project. There is no support for: (i) file permissions; (ii) hard or symbolic-links; or (iii) entities other than traditional files and directories (*e.g.*, no pipes, sockets, *etc*). Adding support for (i) is not difficult and may be done later, whereas support for (ii) and (iii) is more difficult and might be beyond the scope of the challenge. Existing flash-memory file-systems, such as YAFFS2 [55], do not support these features, since they are not usually needed for the kinds of embedded systems with flash memory.

3. Roadmap structure

We present in this paper a roadmap for the *POSIX* pilot project, containing an organised set of activities, interests, documents, goals, and achievements. It has been compiled in 2007, and represents the result of painstaking digging through articles, standards, books, and other sources.

Over this period, in the various workshops and conference meetings, three important milestones came to life: (i) a good synergy of interest with various researchers from different technical background; (ii) enough critical mass of results in the form of models, theories, tools, and experiments; and finally, (iii) requests from different sources wishing to participate, but not knowing where to start, or where to look for information. These led us to write this paper. We tried to organise our achievements, as well as all the work that remains to be done, in the hope that colleagues would go beyond being merely interested and start to get involved in doing the work.

We have divided this paper in five branches, each tackling different aspects of the mini-challenge. After considerable search through documentation, external guidance, and other sources, we decided to follow a guideline for file systems given in an Intel architecture document [29], which allowed us to divide the challenge into three aspects layered orthogonally: (i) file store functionality; (ii) hardware interaction; and (iii) fault tolerance. This architecture follows *POSIX* conventions and is particularly useful for flash-hardware devices, where *Power Loss Recovery* (PLR) is guaranteed. Flash hardware is very relevant to space-flight missions, since the devices have no rotating parts, but the architecture can also be generalised to other types of hardware, as well as other fault-tolerance concerns.

One important benefit of this orthogonality is that it allows different groups applying varied techniques to collaborate while working in parallel. This makes the file store project collaborative, in contrast to the previous Mondex pilot project, which was essentially competitive [16]. Since *POSIX* is a considerably bigger, more copmplex problem, this seems the best way forward. At first, we intend to work on a minimal file system that is enough for NASA's purposes, but the ultimate goal is much wider than this: to achieve conformance certification (see Section 5).

In what follows, we present a roadmap following Intel's architecture. Our presentation of the roadmap does not portray the chronological order in which the documentation was found and the work was built-up. Instead, it provides a logical and systematic view of our researches. For each section, we present the work done so far, what remains to be done, the known participating and interested parties, where the remaining challenges lie, and so on. Our aim in this paper is to distill our findings and the relevant references in a logical and discernible way to inspire others to join us!

4. Intel's architecture

In 2004, Intel released a document containing a set of API's for file systems that is layered at various levels of abstraction [29], and is particularly targeted at flash hardware. It contains the API's signature (in C) with main data structures, error codes, expected functionality, control-flow algorithms, contracts among various layers, and so on. This architecture clearly states a reference guide for file system implementation that carefully considers varied aspects, such as file store functionality, operating system interoperability, real-time issues, hardware interfacing, fault tolerance aspects at various layers, and so on. Unfortunately, to our knowledge, a reference implementation for this architecture is not freely available. Yet, based on the given level of detail, this reference API is most likely to have been derived from some implementation.

Our aim is to try to link our formal models created from the use of the various verification techniques, so that a reference implementation can be constructed correctly through the refinement calculus [39, 2]. We want to have a verified file store that is correct by construction using refinement, from requirements down to code, shaped according Intel's API's, and taking into consideration both fault tolerance and flash hardware aspects along the way.

These API's are laid out in eight layers wrapped up at the top-level to look and behave like the *POSIX* standard API. We divided these layers into four categories, where each

category has a set of important associated documents: (i) file operations involving both the code for a file database, as well as the actual file data itself; (ii) virtual blocks and their operations, which maps physical into local blocks of various sizes providing garbage collection and space management; (iii) device independent and device abstraction operations; and (iv) device-specific operations.

Fault tolerance is handled by different techniques at different layers, where the property under concern is power loss recovery. It is maintained through different transaction processing schemes. For instance, redundant virtual (and physical) areas are used at lower layers closer to hardware.

In what follows, we present a brief top-down description of each layer, addressing both their functionality, their interaction, and the way fault tolerance is added at each level.

Top-level *POSIX* API layer This layer represents the entry point of the execution flow for any file system operation. It assembles the file system layer API's below into standard *POSIX* file system API's [41].

As Intel put it [29, p.12], it is a "concise interface that is *POSIX*-aware." It is used "to implement all file and directory operations per the *POSIX* standard", where "variances are documented and called out (...)", since "certain operating systems have already implemented" parts of these artefacts. An implementation of the standard *POSIX* API's containing all available file system functionality is supposed to wrap up the layered API's described below.

Since different operating systems already have their own file system requirements, and their own interfaces for varied resources, an intermediate OS-dependent layer is needed in order to accommodate the general file system core.

Real-time OS-wrapping and OS-resources translation layers The real-time OS wrapping layer provides an entry-point for operating system specific API's, hence it is not described in the Intel document. The description of the layer allows room for OS-specific implementation needs. Similarly, the OS-resources translation layer provides an access point to required resources managed by the operating system, such as mutex semaphores, background threads, hardware interrupts, error codes, and so forth.

Calls to API's in these layers from within the main file system layers means that allocated resources at the different layers are allocated and released properly. For example, when a flash array (*i.e.*, the flash-hardware abstraction within the low-level layer) is no longer needed because the volume is unmounted, the device driver will be released; or when an error is returned by the Intel API's, it is translated into the appropriate operating system error code, and control flow is transferred to the right thread of execution.

These OS-wrapping API's are also responsible for telling the file system layer if transaction processing is required or not, and that appropriate permission checks have not failed for the requested operation within the operating system access control policies.

File system layer This is a generic interface into the file system that understands the *POSIX* conventions, and which is used by individual operating systems to communicate with the file system core. The requirements of this layer are pretty much determined by the *POSIX* standard specification. The later handles file system operations through twenty available API's, where calls are directed down to the appropriate data object API. That involves: (i) locating static file information, as well as the file data root; (ii) performing the file system operation over the right data object; (iii) updating both static and dynamic meta-data file information, in the case of write-like operations; (iv) returning appropriate results and/or error codes; and so on. It also provides file and directory management in order to keep track of open file handles within the file system that are being used by the operating system.

Fault tolerance for PLR is added at this level by using a transaction processing mechanism within the file database in the data object layer below, so that multiple operations per file handle are allowed until a commit or rollback takes place. This is much like what was implemented for the IBM CICS transaction processing system [24, 23], which has some mechanised models available [14, 38, 56, 18], and is further discussed below (see Section 6).

The layer also provides other user facilities like: searching, moving, and renaming of files and directories; file access, permission, and cache control; volume management for partitioning and user quotas; multi-threaded access; custom set of API's (packed within one API) for non-*POSIX* functionality; and so on. Together with the twenty API's, there are eighty different return/error codes related to varied aspects of the functionality, such as general file system behaviour (21), specific flash file system behaviour (34), or the interaction with the data object layer (25). Seven complex data structures are defined, and they handle: (i) a file allocation table; (ii) volume format; (iii) space usage information, such as fragmentation size granularity and allowance; (iv) various initialisation options, such as maximum number of open files or path length; (v) types of file; (vi) file and directory information, such as size, attributes, and time-stamp; and (vii) storage/caching for open file information like seek offset, and access (*e.g.*, read-only, read-write, *etc*) and share (*e.g.*, share-read, share-write, *etc*) modes. Together with file access and share modes, there are other execution modes also defined for: file search, open, and seek; RAM buffer usage; transaction operation status; and non-*POSIX* I/O control sub-API commands.

Data objects layer It provides a common structural organisation for different types of data, such as static and dynamic allocation information (*e.g.*, file access table), file data and directory pages, *etc*. In this way, it offers an interface with 23 API's for the file system components to uniformly access and manipulate data. Among other functionalities, it also: provides a uniform way of manipulating these various types of data; joins together logical units of data separate across multiple virtual blocks, which is important to handle fragmented data or larger devices; accesses appropriate logical units irrespective of their type; and so forth. These enable the translation of read/write commands within varied types of data into read/write commands within multiple uniform logical units of data.

Power Loss Recovery is added at this level through partial data write schemes. This allows transacted operations on a file, where appropriate PLR steps are taken whilst writing the data, so that all partial data writes are successful. All layers have initialisation and shutdown API's to detect any power loss issues (or indeed other fault tolerance characteristics needed), and to perform the appropriate recovery operations. Not surprisingly, each layer calls the layer below in order to perform lower-level initialisation/shutdown.

To implement the PLR schemes, a set of extra PLR-specific API's is also provided (*i.e.*, 10 of the 23 API's), where failure error codes provide the guarantee that no change is made to persistent data.

Basic allocation layer It breaks up physical erase blocks into logical units of equal size, whilst keeping track of individual units and their status. It also keeps track of the used, free, and dirty (invalid/erasable) space within each erase block. In this way, it manages available space within a given volume, which includes the functionality needed for volume formatting, mounting, and unmounting. One well-known file system part within this layer is the file allocation table, which is built in RAM.

It allocates units on request from the data object layer above, one unit at a time (*i.e.*, one execution thread at a time), and is responsible for PLR of each logical unit. It is indifferent to the type of data requested, since the data object layer above performs requests in logical block units. The way PLR schemes are implemented at this level depends on the nature of the hardware being used. The specific details on how PLR is performed at the logical (virtual) units level are similar to the way it takes place at the physical (hardware) block level, where a translation layer between logical and physical blocks/volumes is in place.

A series of logical/physical tables are managed and maintained by this layer, which together with the reclaim layer, are the most complex in the whole architecture design. For instance, for flash hardware, the logical block table keeps track of the various wear-levelling schemes needed to avoid early malfunction of flash hardware.

Requests are then passed to the flash interface layer below. By doing so, basic allocation can take place regardless of the kind of device being used. During initialisation, the flash interface layer is called again in order to detect (or build) a physical block table. There is also an interaction with the reclaim layer, which is placed alongside the basic allocation layer. Reclaim (or free-space garbage collection) is performed by associating logical block numbers to each physical erase block within the volume. Basic allocation then divides and allocates particular units of equal and predetermined size for the data object layer above.

During reclamation, a backup mechanism is in place in case of power loss; the recovery process restores physical block information that would otherwise be lost. Error detection and correction (*EDAC*) schemes are also incorporated, so that the file system can detect and recover from corruption to the file system structure. For that, redundant meta-data is physically stored.

This is certainly the most complex of all layers. It is further divided into seven sub-layers, which we have separated in three categories: (i) interfacing with the data objects layer above, and the flash interface below; (ii) logical-to-physical block mapping and management, which is a quite delicate task, as it can severely affect the file system's performance; and (iii) physical block management, which includes compression and *EDAC* algorithms. Among these layers there are around 15 API's, 9 complex data structures representing the various mappings.Again, failure error codes ensure no change has been made to persistent data.

Reclaim layer It is used for garbage collecting free space from the dirty (invalid) space generated during file system operations. It copies data from source (dirty) to destination (free) blocks, so that the source block is reclaimed as free space for future write operations. This includes preemptive reclaim before large file writes. These processes are completely shielded from power loss. The employed PLR mechanisms are also further protected with *EDAC* algorithms. Moreover, it is here that the wear-levelling algorithms used in flash hardware reside.

An erase block is reclaimed based on the amount of dirty space it contains, which depends on the frequency that the logical block is deleted or updated. As usage varies widely among applications, it is impossible for the basic allocation layer to optimally foresee what is to be reclaimed. Information on free/dirty space is fed back to the basic allocation layer to maintain the logical block information structure.

During reclamation, each physical erase block of a volume is assigned a unique logical block address that is stored within the block itself, except for the destination block, which will store data. Then, upon reclamation of a physical block, its logical block address is copied along with

valid data to the destination block as part of the reclamation process. Thus, the logical address "follows" the data of the erase block it is being moved from. When the basic allocation layer accesses the reclaim information, it must perform logical-to-physical block address translation. This is a known bottleneck of the file system, since it is frequently performed, and should be quite carefully implemented to achieve optimal performance.

As reclamation is power-loss recoverable, the process of copying valid data to some other free location before erasing data must be carefully performed. A naïve approach would be to use the basic allocation RAM cache. This is a bad idea for two reasons: (i) erase blocks tend to be large, hence requiring much memory; and (ii) in the event of power loss, all data in transit on the RAM would be lost. Instead, this layer sets aside one logical block (*i.e.*, the destination block) to be used as temporary storage during the reclaim process. This way, the reclaim destination block is never used to store data, but only for the reclaim process. When a source block is targeted for reclaim, the destination block is erased and contains no data. Next, only valid data is copied to the destination block at the right offsets. After all valid data is copied, the original source block becomes the next spare destination block. Also, a *Reclaim-In-Progress* (RIP) write-flag is used per block to differentiate between a block being written to a block being reclaimed. A table of RIP flags is maintained by the basic allocation layer.

For flash hardware, the reclaim module performs another fundamental task: *wear-levelling*. As flash physically degrades on writing, the reclaim layer also ensures that adequate wear-levelling of physical blocks is in place in order to maximise the flash usage life.

Due to its complexity, more information on the use of the eight reclamation API's is also available. These are five specific state-machine-like algorithms on how to select, idle, relocate, erase, and update different source/destination blocks during reclamation, which includes handling the RIP flag. External OS-resources are required, such as mutex semaphores, and background daemon threads. Semaphores are used to guarantee that only one reclaim per volume can occur at a time, since there is only one spare destination block per volume to perform reclamation; whereas a background thread is kept idle until the semaphore-protected erase region is available for reclamation.

As both basic allocation and the reclamation process seem to be amongst the most sensitive and complex tasks within the file system core, a great degree of care ought to be taken when implementing them. For instance, as they are cental to file store performance, one challenging extension would be to have multi-threaded reclamation algorithms, which also consider wear-levelling of the flash with EDAC of data in the process. In a survey of algorithms and data structures for flash memories [21], various options are explained, yet none exploit concurrency. The same is true for patented industrial wear-levelling algorithms (*e.g.*, `www.patentstorm.us/patents/6732221-description.html`). This is an exciting area for research with direct industry impact and interest.

Flash interface layer It shields upper layers from platform-specific low-level device drivers. Its five API's are quite simple: `read`, `write`, `erase`, `initialise`, and `shutdown`. They are used to translate all basic allocation and reclaim requests into simple read, write, or erase calls that are platform/hardware specific. They communicate with the appropriate device driver, hence linking the file system with a particular hardware device. With 31 additional execution (result/error) codes, this layer reports back to the file system information on the managed hardware, such as block size and block count. These return codes are then translated back into appropriate file system error codes.

For flash hardware, this layer translates file system volumes into flash arrays, which are an abstract view of the way the physical device is laid out (see Section 7). For file system operations, it translates the basic allocation layer operations into physical flash device operations, such as page programming or physical block erasure [26, 5, 6].

Any system resources obtained during these hardware-related operations are released. For instance, when a volume is mounted, the corresponding device driver will be loaded. That also means that initialisation and shutdown of hardware, which may include hardware PLR and other fault tolerance schemes, are performed.

Low-level hardware layer This represents the device driver directly manipulating the hardware. To ensure integrity, this is the only available access point to hardware. Depending on the platform and device type, the driver behaviour (and implementation) may vary.

Hardware interaction obviously depends on the device. For flash, there is usually the ability to perform read-while-write operations, where specific quality-of-service timing constraints are in place [26, Ch. 4]. For instance, it is possible for some devices to erase a physical block, while executing code from another physical block in the same device.

Such operations are performed either from the hardware itself or via software. For hardware read-while-write, disabled interrupts and scheduling ensure proper completion of the command cycles [26, Ch. 7], whereas software read-while-write operations also work with interrupts and scheduling, but uses flags indicating the mode the flash array is in, to control operation suspension and pool for hardware interrupts requests.

Other functionalities encompass: creating virtual flash devices to allow a single physical device to be broken into

multiple virtual devices, and vice-versa (*i.e.*, multiple physical devices into single virtual device); interleaving flash devices across a data bus for improved data throughput; striping data across multiple (virtual) devices, hence allowing multiple writes to take place concurrently; physical block locking; *etc*. These features are particularly useful to assemble large amounts of data, and to considerably increase device's performance.

The nine low-level API's operate over nine general flash-device structured data types directly linked with the kind of hardware being handled, where 26 error codes specific to flash are used to provide detailed information back to the flash interface layer above. For each API, a detailed control-flow algorithm and finite state machine is given.

As Intel is involved in standardising flash hardware, we expect to formally relate the finite state machines from [26, Ch. 7], with the algorithms and control-flow diagrams provided in the architecture document. For instance, one option would be to use process algebras like CSP [43] to analyse these information/control flow diagrams. One successful story of such attempt is an automated technique developed in Brazil and used by Motorola, which goes from a requirements table, through state machine diagrams, down to a CSP process that can be model checked [7].

5. *POSIX* and the Open Group

POSIX is the *Portable Operating System Interface*: an open operating system standard interface produced by IEEE that is recognised by both ISO and ANSI. It is widely accepted word-wide, with UNIX and Linux being the best-known implementations (see `www.unix.org`).

A reference implementation can be either *POSIX* conforming or *POSIX* compliant. The former means the implementation adheres in full to the published standard, where various optional subsets may also be included, such as threads or real-time extensions. The latter means the standard is partially adhered to, where documentation must be available showing which features are supported. Another terminology used is *POSIX*-aware, which means that although an implementation does not conform (or comply) to *POSIX*, it uses some known *POSIX* ideas or API's. Certification for conformance is granted by accredited and independent certification authorities, and is managed by IEEE (`get.posixcertified.ieee.org`) and the *The Open Group*. The latter is the "vendor-neutral and technology-neutral" consortium responsible for "developing a range of services that provides strategy, management, innovation, standards, certification and test development". It was through them that we first got in touch with an invaluable source for *POSIX* [33].

The certification process is quite lengthy, and various stages are involved. There are also different levels of certification. The whole of *POSIX* is formed by around 1,800 API's with over 4,000 pages of documents. The process for the current IEEE 1003.1-2003 version involves a quite impressive test-suite (see `get.posixcertified.ieee.org/docs/testsuites.html`) with around 23,000 test cases, which need to succeed in under 12 hours of execution. Certification can take up to eight months, yet it usually takes two, and is valid for twelve months. At present, the certification process costs between US$5k–18k, depending on various factors, such as being a new certification or a renewal, being a product family or platform-specific certification, and so on.

During the early phase of the standardisation process, the *POSIX* API's requirements [28] (done in July 1995) were formally specified in [42] using the Z notation [47] (in August 1995). That is, an abstract specification capturing these requirements in Z was created from the set of informal requirements to show how they were sometimes ambiguous or contradictory. This work is the model for the top-level *POSIX* API wrapping-up layer discussed above. This Z specification served as a guideline for the actual version of the *POSIX* standard (from February 1998), with a revised version published in 2003 [33]. In fact, the Z specification from 1995 [42] was inspired by an earlier, even more abstract formal definition of UNIX file systems given in [40]. We mechanised [42] finding some interesting results. Due to lack of space, we will not discuss this further; some of the results are available in [20].

This was the starting point of our bibliography archaeology work and it took place in the end of 2006. In the remainder of this section we present our results on this front.

UNIX filing system In [40], Morgan & Sufrin described an abstract specification of a UNIX filing system given by few data structures and operations of files and file storage using the Z notation [47] (the *M&S* specification). It is divided into three parts comprising: (i) file storage database with file creation and data manipulation operations; (ii) file descriptors (or channels) used for random access of file data (*i.e.*, a file access table); and (iii) directories and links. Although it does not completely model *POSIX* behaviour, such as error codes or file permissions.

This was our starting point for the functional requirements for the subset of *POSIX* that is within JPL's interest. As the *POSIX* standardisation body chose the Z notation, and we are quite familiar with Z and its tools, we followed suit and started the work in Z as well. Nevertheless, this is not necessarily a requirement of the project, and other formalism such as B [45] could be used as well.

We took the file database and the directories of the *M&S* specification and mechanised them using the Z/Eves theorem prover [44]. That meant parsing, typechecking, consistency checking (*e.g.*, making sure that partial functions

are applied within their domains), adding automation lemmas, and so on. In this process alone, we found (and fixed) some inconsistencies from the original *M&S* work. This resulted in two MSc. theses [19, 30]. For the file database, we proved a data refinement to a Z hash map of the filing operations using an extended set of forward simulation laws [51, Ch. 16-18], which include refinement of the API interface [8]. Hash maps were considered, as it we have evidence that industry-scale file systems use such data structure. This resulted in a series of publications [15, 13, 17].

Since a Z hash map is a result in itself, and since the Verified Software Repository is as much about verified components as verified experiments, we did some extra work on exporting (informally, but systematically) some of the Z hash map invariants and operation pre and postconditions as JML [3] annotations for Java `HashMaps` [15]. These annotations can be used to perform further checks on the Java code, such as static analysis with ESC/Java [9], or functional correctness verification with JACK [4], loop/invariant detection with LOOP [25], and so on. This turned out to be helpful in prototyping a flash file store for Java. The exercise was also useful in foreseing a possible link between Z and JML, as well as extending the JML mathematical type system with the Z mathematical toolkit.

And there is still plenty of work to do. We would like to refine both the file seeking and directory structure down to code in a similar fashion. There is ongoing work in describing the directory structure part of *M&S* using Event-B and the RODIN tools (`www.event-b.org`). We also plan to refine it to a B^+-tree specification [10] written in VDM [31], which we think is another quite interesting exercise. This B^+-tree specification shows how to go from an abstract VDM B^+-tree down to an annotated *Pascal* code using refinement.

Recently, we worked on theoretical results on how to trade theorems among different theories [53], and in this case, different logics: from the three-valued logic of VDM, to the semi-classical two-valued logic of Z. With that and some other minor adjustments, we believe it is possible to use Z tools to mechanise this specification and generate code annotations for the B^+-tree in a similar way we did for the Z hash map mentioned above, in order to allow code-level verification. This time, we want to generate annotations for C# using the Spec# tools (`research.microsoft.com/specsharp`). In a recent visit to *Microsoft Research* (Redmond), in order to familiarise ourselves with Spec# and its tools, it turned out that such a B^+-tree could be of interest for the *Vienna Hypervisor*: the hardware abstraction layer written in C that *Vienna*, *Microsoft*'s next operating system, will use for virtualisation.

Other interesting aspects of the problem yet to be tackled are: (i) introduction of fault tolerance aspects to the file store database; (ii) reference implementation for the relevant Intel API layers that has been derived through refinement from the requirements, where formal annotations are added and analysed in the source code; (iii) test-case generation and model based testing, as advocated in [50]; and so on.

6. IBM CICS and fault tolerance

We mechanised the Z specifications of two modules of the IBM CICS transaction processing system. They were the file control [24] and task control [23] API's [14, 38, 56, 18]. They are important because they enabled us to inject fault tolerance for files, as well as the conditions under which API's can be called. There is considerable work to be done in porting the results from IBM CICS into Intel's API's. Due to lack of space, we do no discuss this further.

7. Open NAND Flash Interface (ONFi)

Flash memory devices are often used as the data-storage medium of choice. Of particular interest are file stores based on the relatively recent NAND flash Memory technology, which has a recent standard [26]. NAND flash is now very popular in portable devices, such as MP3 players and datakeys. Flash memory is seen as ideal for these purposes as it has good physical handling properties: it is non-volatile, shock-resistant, and capable of operating under a wide range of pressures and temperatures. For spacecraft, it is even more valuable since it has no moving/rotating parts.

There are two types of flash memory: (i) NOR flash memory, which can be programmed (written) at byte level, but must be erased at block level, is relatively slow, but suits random access; and (ii) NAND flash memory with higher speed, but where programming must be done at the page level, making it a sequential access device. The former suits non-volatile core-memory, whilst the latter is better suited for implementing file systems.

A key issue when in developing these devices is in the way the hardware is physically laid out, as well as its available features, which usually varies by manufacturer. This difficulty is then inherited by those designing equipment relying on flash memory, which was a limiting factor in the adoption of the technology.

In order to alleviate this situation, leaders in the flash manufacturing industry formed a standardisation organisation: the *Open NAND Flash Interface* consortium (ONFi). Their aim is to develop a common standard to which most manufacturers would then adhere. The current ONFi standard is then the obvious choice as our modelling target.

Flash device data modelling A flash memory device is decomposed hierarchically into targets, logical units, blocks, pages, and data units. Each of these are important

for modelling different perspectives, as they capture boundaries that define if and when operations can be performed.

The basic data unit in a flash memory is either a `Byte` (8 bits), or a `Word` (16 bits), depending on the type of device. A page is an array of data items, consisting of a main page, plus some spare locations used for *EDAC*. A page is the basic unit for programming (writing). A block is a collection of pages, and is the smallest unit to which an erase operation can be applied. A logical unit (LUN) is the smallest sub-entity within a device that is capable of operating independently. They compare to the logical blocks from Intel's basic allocation layer. It comprises a collection of blocks, along with a status register and at least one page register in RAM. A target, within a device, is the smallest unit that can communicate independently off-chip. It is made of one or more logical units.

An erased data item has all its bits set to logical 1's, and programming a data item involves changing some of those to 0's, which means that overwriting already written data is typically not possible without an intervening erasure operation. Erase operations need to be kept as low as possible, since flash memory physically degrades when erased.

Such formal models of flash devices capture the relevant aspects of behaviour (*i.e.*, functional, performance/timing, reliability, *etc*), in a way that allows them to be tied into formal descriptions of the surrounding hardware and associated software operations.

Flash device chip-set operations With the data structure for such devices modelled, we need to provide support for key operations, such as programming, reading, and erasing, as well as other fault-tolerance features like PLR. These are specified as a command set table with 22 operations, where 13 are optional.

This layered modelling of ONFi devices is quite interesting, as consistency checks can be performed at different levels of abstraction, as shown in recent publications [5, 6]. This result can be further exploited by industry to explore different operation configurations.

For instance, each operation can be modelled as atomic, hence the flash is performing one operation at a time. Still, current flash device are not atomic and require the appropriate sequencing of inputs and outputs to complete any operation, as well as the need to wait for certain tasks (typically data transfer) to complete. This provides an opportunity to optimise performance by interleaving operations and the use of cache techniques. Most of those are part of the optional operations within the ONFi standard. All these options are described to some level of detail in the ONFi standard, and an obvious modelling goal is the formal description of all three, as well as how they relate.

No matter the level, a key issue that arises is that certain operations may fail, with various degrees of observability. The probability of failure is initially very low, but rises over time, as measured by the number of operations performed. This requires wear-levelling algorithms at the hardware-level to minimise the failure rate. This requires us to model failure properly, with a particular emphasis on the fact that such failures have a persistent and lasting effect.

Our initial model of ONFi [5, 6] focuses on the first level, viewing operations as atomic. A key concern was to describe formally the state in which devices are shipped, as memory faults will already be present as bad blocks, and there is a scheme in place to mark such blocks.

There is no other published work on the formal modelling of NAND flash devices, to the best of our knowledge, but there has been a considerable body of work done on formal models of file systems, and the technical, usage, and reliability aspects of NAND flash devices. We know that work with the ONFi standard is also being undertaken at the University of Minho (in Portugal), which is led by José Nuno Oliveira, but no publication is yet available.

Flash device finite state machines The internal control of flash devices relies (conceptually at least) on a collection of communicating finite state machines, whose interactions support the sequencing and interleaving of operations. These state machines are quite similar to those presented in the low-level layer in Section 4.

A wide range of material has been published regarding the implementation of file systems on NAND flash memory, most of which utilise some form of log-structuring [32, 54, 55, 36]. Of interest to a potential space application are techniques that use NAND flash to implement low-power file caches for mobile devices [37, 35]. A key feature is the need to cope with the accumulation of errors over time, a mechanism which is well understood [1, 46].

Continued standardisation The ONFi standard is evolving continuously, with version 2 due out in early 2008. The ONFi consortium have another standard [27] that looks at devices capable of supporting, at the hardware level, an access mode based on logical block addresses, which are always 40 bits in length, accessing blocks of 512-bytes, regardless of the underlying block size or number of address bits of the real device.

A separate committee, entitled *Non-Volatile Memory Host Controller Interface* (NVMHCI), is another standardisation initiative on flash device drivers lead by Intel, which works alongside ONFi. Its first standard draft should be ready by early 2008. It corresponds to the functionality described in the flash interface layer API.

The University of York has become the first academic member of ONFi, and we are currently discussing NVMHCI membership as well. Membership entitles early

access to non-public draft documents, as well as participation in various moderated discussion lists. Once we better understand the underlying models, we intend to present our findings to the standardisation committee through these discussion lists. In a recent Grand Challenge workshop at ICFEM in Florida (in November 2007), members of ONFi participated and gave interesting insights on the issues they face. We also had the chance to show some inconsistencies in the current public standard and ask advice on how to proceed, as well as clarifying some hidden assumptions within the standard.

8. Conclusions

In the quest to formally specify a *POSIX* file store, we divided the work suggested in [34] following an orthogonal architecture provided in [29] that enables separation and later combination of concerns, such as functional requirements, fault-tolerant imperatives, and various hardware devices. This is crucially important in order to allow collaborative work among scientists with different interests and backgrounds to collaborate in completing the challenge.

We report on the various references we found, and how they all fit together to make a cohesive and comprehensive body of work. We defined a roadmap reporting achieved goals, summarising important information, and offering suggestion for collaboration as "micro-challenges" within the pilot project. Within the roadmap, we identified opportunities and presented results that contribute to all three branches of the verified software verification Grand Challenge: theories, tools, and experiments.

Our efforts are firstly aimed at a particular user (*NASA*'s JPL), hence we concentrate on an initial small subset of *POSIX* file store functionality of their interest. This is possible because we follow *Intel*'s architecture mentioned above, hence we have a modular project development strategy. The results of this work are collected and available on-line at the VSR repository at SourceForge [11].

Collaboration and exploitation In due time, this effort could lead to a formally verified *POSIX* compliant file store that is widely used in other main stream industries. In this process, undoubtedly new tools will be created, mature tools will be improved, theories will be extended, and other experiments will benefit from the results.

Future work Throughout the document we pointed out various opportunities for collaboration and ambitious "micro-challenges" within the *POSIX* verification Grand Challenge pilot project. We are working on an extended version of this roadmap, as well as a commented bibliography paper providing a reference manual for the various documents and sources available.

References

[1] S. Aritome et al. Reliability issues of flash memory cells (invited paper). *Proc. of the IEEE*, 81(5):776–788, May 1993.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Text in Computer Science. Springer-Verlag, 1998.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll5. An Overview of JML Tools and Applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Electronic Notes in Theoretical Computer Science, pages 73–89. University of Nijmegen, Elsevier, March 2003.

[4] L. Burdy, A. Requet, and J. L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Proceedings of Formal Methods Europe, Pisa*, number 2805 in Lecture Notes in Computer Science, pages 422–439. Formal Methods Europe, Springer-Verlag, 2003.

[5] A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a Formal Model of Flash Memory. *Science of Computer Programming*, 2008. under review.

[6] A. Butterfield and J. Woodcock. Formalising flash memory: first steps. In 12^{th} *ICECCS*, pages 251–260, Auckland, Jul. 2007. IEEE.

[7] G. Cabral and A. Sampaio. Formal specification generation from requirement documents. In *Brazilian Symposium on Formal Methods (SBMF)*, 2006.

[8] D. Cooper, S. Stepney, and J. Woodcock. Derivation of Z Refinement Proof Rules: Forwards and backwards rules incorporating input/output refinement. Technical Report YCS-2002-347, University of York, 2002.

[9] D. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center (SRC), www.research.digital.com/SRC/, 1998.

[10] E. Fielding. The specification of abstract mappings and their implementations as B^+-trees. Technical Report PRG-18, Oxford University, 1980.

[11] L. Freitas et al. Verified Software Repository @ SourceForge. http://vsr.sourceforge.net/gc6index.html, 2006.

[12] L. Freitas et al. Workshop on the vsr grand challenge: *POSIX* file stores. Dublin, 2006.

[13] L. Freitas, Z. Fu, and J. Woodcock. *POSIX* file store in **Z/Eves**: an experiment in the verified software repository. In 12^{th} *ICECCS*, Auckland New Zealand, Jul. 2007. IEEE.

[14] L. Freitas, K. Mokos, and J. Woodcock. Verifying the CICS File Control API with Z/Eves: an Experiment in the Verified Software Repository. In 12^{th} *ICECCS*, pages 290–298, Auckland New Zealand, Jul. 2007. IEEE.

[15] L. Freitas and J. Woodcock. Proving Theorems about JML Classes. In *Formal Methods and Hybrid Real-time Systems*, volume 4700 of *LNCS*, pages 255–279. Springer, 2007.

[16] L. Freitas and J. Woodcock. Mechanising Mondex with Z/Eves. *Formal Aspects of Computing Journal*, 20(1), January 2008.

[17] L. Freitas, J. Woodcock, and Z. Fu. *POSIX* file store in **Z/Eves**: an experiment in the verified software repository. *Science of Computer Programming*, 2008. under review.

[18] L. Freitas, J. Woodcock, and Y. Zhang. Verifying the CICS File Control API with Z/Eves: an experiment in the verified software repository. *Science of Computer Programming*, 2008. submitted for revision.

[19] Z. Fu. A refinement of the UNIX Filing System using **Z/Eves**. Master's thesis, University of York, Oct. 2006.

[20] Z. Fu. POSIX 1003.21 Standard in Z/Eves. Master's thesis, University of York, Sep. 2007.

[21] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computer Surveys*, 37(2):138–163, 2005.

[22] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[23] I. Houston. The CICS Application Programming Interface: Task Control. Technical Report TR12.307, IBM UK, Hursley Park, 1991.

[24] I. S. C. Houston and J. B. Wordsworth. A Z Specification of Part of the CICS File Control API. Technical Report TR12.272, IBM UK, Hursley Park, 1990.

[25] M. Huisman. *Reasoning about Java Programs in Higher-Order Logic using PVS and Isabelle*. PhD thesis, Universiteit Nijmegen, 2001.

[26] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0, ONFI, www.onfi.org, Dec. 2006.

[27] Hynix Semiconductor et al. Open NAND Flash Interface Specification: Block Abstracted NAND. Technical Report Revision 1.0, ONFI, www.onfi.org, 18th July 2007.

[28] IEEE POSIX Working Group. Interface Requirements for Realtime Distributed Systems Communication. Technical Report IEEE P1003.21, IEEE, Jul. 1995.

[29] Intel Flash File System Core Reference Guide, version 1. Technical Report 304436001, Intel Coorporation, Oct. 2004.

[30] V. S. Jegannathan. Specification and Refinement of a Naming System in Z, for the UNIX File System. Master's thesis, University of York, Sep. 2007.

[31] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2^{nd} edition, April 1990.

[32] H. joon Kim and S. goo Lee. A new flash memory management for flash storage system. In *COMPSAC*, page 284. IEEE Computer Society, 1999.

[33] A. Josey, editor. *The Single UNIX Specification Version 3*. Open Group, 2004. ISBN: 193162447X.

[34] R. Joshi and G. J. Holzmann. A Mini-Challenge: Build A Verifiable Filesystem. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, Zurich, Switzerlan, 2005. IFIP Working Conference.

[35] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, New York, NY, USA, 2006. ACM Press.

[36] S.-H. Lim and K.-H. Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, July 2006.

[37] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In T. N. Mudge and B. D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences, Vol. I: Architecture, HICSS'94 (Maui, Hawaii, January 4-7, 1994)*, volume 1, pages 451–460, Los Alamitos-Washington-Brussels-Tokyo, 1994. IEEE Computer Society Press.

[38] K. Mokos. Specifying the IBM CICS File Control API For the Verified Software Repository. Master's thesis, University of York, Sep. 2006.

[39] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[40] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. *IEEE Transactions on Software Engineering*, 10(2):128–142, 1984.

[41] Open Group Technical Standard. Protocols for Interworking: XNFS, Version 3W. Technical Report C702, The Open Group, Feb. 1998. ISBN: 1859121845.

[42] P. Place. POSIX 1003.21—Real Time Distributed Systems Communication. Technical report, Software Engineering Institute @ Carnegie Mellon University, Aug. 1995.

[43] A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[44] M. Saaltink. *Z/Eves 2.0 User's Guide*. ORA Canada, 1999. TR-99-5493-06a.

[45] S. Schneider. *The B-Method—an Introduction*. Palgrave, 2002.

[46] A. Sikora, F.-P. Pesl, W. Unger, and U. Paschen. Technologies and reliability of modern embedded flash cells. *Microelectronics Reliability*, 46(12):1980–2005, 2006.

[47] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1998.

[48] SRI. Workshop on the Verification Grand Challenge. www.csl.sri.com/users/shankar/VGC05, Feb. 2005.

[49] S. Stepney et al. An Electronic Purse: Specification, Refinement, and Proof. PRG 126, Oxford University, Jul. 2000.

[50] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 1st edition, 2007.

[51] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, 1996.

[52] J. Woodcock and L. Freitas. **Z/Eves** and the Mondex Electronic Purse. In 3^{rd} *ICTAC*, volume 4281 of *LNCS*, pages 15–34. Springer, 2006.

[53] J. Woodcock and L. Freitas. Linking VDM and Z. In *In Proceedings of* 13^{th} *ICECCS, Belfast*, LNCS. Springer, 2008. under review.

[54] D. Woodhouse. JFFS: The Journalling Flash File System. *Ottawa Linux Symposium*, Oct 2001.

[55] YAFFS Direct Interface (YDI) User's Guide. www.aleph1.co.uk/node/349, Jul. 2006.

[56] Y. Zhang. Specifying the IBM CICS Task Control API For the Verified Software Repository. Master's thesis, University of York, Sep. 2007.