Model-Checking the Linux Virtual File System*

Andy Galloway[1], Gerald Lüttgen[1], Jan Tobias Mühlberg[1], and Radu I. Siminiceanu[2]

[1] Department of Computer Science, University of York, York YO10 5DD, UK.
{andyg,luettgen,muehlber}@cs.york.ac.uk
[2] National Institute of Aerospace, Hampton, Virginia 23666-6147, USA.
radu@nianet.org

Abstract. This paper presents a case study in modelling and verifying the Linux Virtual File System (VFS). Our work is set in the context of Hoare's verification grand challenge and, in particular, Joshi and Holzmann's mini-challenge to build a verifiable file system. The aim of the study is to assess the viability of retrospective verification of a VFS implementation using model-checking technology. We show how to extract an executable model of the Linux VFS implementation, validate the model by employing the simulation capabilities of SPIN, and analyse it for adherence to data integrity constraints and deadlock freedom using the SMART model checker.

1 Introduction

Hoare has proposed a 15-year grand challenge which calls on the program verification community to collaborate on building verifiable programs [16]. Joshi and Holzmann have subsequently provided a mini-challenge [19] of building a verifiable file system as a stepping stone towards meeting Hoare's challenge. Neither challenge overly constrains the verification approach. On the one hand, for example, there is the *constructive* approach in which formal reasoning is employed to first establish the validity of a specification and then the correctness of an implementation with respect to the specification. On the other hand, the *analytical* approach aims to build a valid abstract model of an existing implementation and then to show that this model satisfies some correctness criteria.

This paper applies the analytical approach to verifying an implementation of the *Virtual File System* (VFS) layer [4] within the Linux kernel, using model-checking technology. This layer is of particular interest since it provides support for implementing concrete file systems such as EXT3 and ReiserFS, and encapsulates the details on top of which C POSIX libraries are defined; such libraries in turn provide functions, e.g., *open* and *remove*, that facilitate file access (cf. Sec. 2). The aim of our case study is to assess the feasibility of analytical program verification to Joshi and Holzmann's mini-challenge. In particular, we

* This research was partially funded by EPSRC under grant GR/S86211/01 and by NASA under cooperative agreement NCC-1-02043.

are interested in whether and how an appropriate model of the VFS implementation can be constructed, and if meaningful verification results can be obtained given the limitations of current model-checking technology.

Our first contribution is in modelling the complex Linux VFS implementation that consists of more than 65k lines of C code, based on an analysis of the data structures and algorithms employed in VFS (cf. Sec. 3). Despite the recent advances in software model checking — as exemplified by the BLAST model checker [15] and Microsoft's Static Device Verifier (*www.microsoft.com/whdc/devtools/tools/SDV.mspx*) which, under the hood, automatically extract models from C code —, one quickly reaches their limits when applying them to operating systems code, such as the VFS code. This is because such code makes use of dynamic memory allocation, function pointers, macros and inlined assembly [21]. Until techniques addressing these shortcomings have matured, building models from operating systems code remains largely a manual task.

Our VFS model is the result of slicing away and abstracting some details of the VFS data structures and algorithms. This is done in a way that makes the model amenable to modern model checkers while maintaining all details necessary for checking non-trivial data-integrity properties. The model is expressed abstractly in a subset of C, so that it can easily be reused by others. While building the model took several weeks, its validation via reviewing and simulation consumed several months. The simulation was carried out in the SPIN model checker [17] since SPIN has rich simulation capabilities, with support for run-time assertions, and an input language into which our model can be cast straightforwardly. However, since our model is sufficiently close to the VFS implementation and thus exhibits a large state space with wide state vectors, we were unable to run SPIN in verification mode, even when disallowing concurrent access to VFS functions. Also, the VFS model cannot be verified by model checkers that do not support concurrency, such as BLAST.

The paper's second contribution is the formal verification of our VFS model by using model checking to analyse low-probability scenarios, thereby increasing confidence in the correctness of the Linux kernel (cf. Sec. 4). In particular, we were looking for, and not expected to find, the corruption of the underlying data state and deadlocks. The challenge here is to identify data-integrity properties from the rather shallow VFS documentation. To conduct the verification, we translated our model into Petri nets and used the model checker SMART [6] which implements efficient, decision-diagram-based algorithms for analysing concurrent systems. SMART was chosen here because of our familiarity with the tool and its proven record for analysing complex models, including NASA's Runway Safety Monitor [22] and the SPIDER clock synchronisation and self-stabilisation protocols [20]. While the VFS model pushes SMART to its limits, we were able to successfully prove all considered properties.

Our case study is novel because of its approach and scope. It tests the feasibility of reverse-engineering a model of an existing file system, including data structures and locking mechanisms, and of checking such a model for adherence to healthiness properties. This contrasts with other work in the field (cf. Sec. 5)

which employs either the constructive approach to verification [2, 11], or model-checking as a run-time verification technique for driving a test-harness for the implementation [23, 24]. Our file system model is of particular interest to NASA which is currently developing, together with JPL, a pilot project to help build a reliable file system for flash memory.

2 The Linux Virtual File System

This section introduces the Linux file system architecture and, in particular, the Virtual File System layer. For a more detailed description, we refer the reader to [4] and *www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html*. An overview of the VFS internals and data structures is presented in Fig. 1

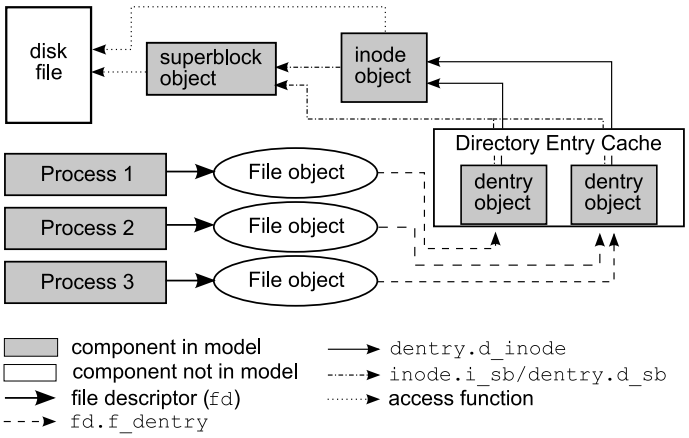

Fig. 1. Illustration of the VFS environment and data structures.

Architecture. The Linux file system architecture consists of six layers. The most abstract is the *application* layer which refers to the user programs; this is shown as Process 1 to 3 in Fig. 1. Its functionality is constructed on top of the file access mechanisms offered by the *C POSIX library*, which provides functions facilitating file access as defined by the POSIX Standard [14], e.g., open `open()`, delete `remove()`, make directory `mkdir()` and remove directory `rmdir()`. The next lower layer is the *system call interface* which propagates requests for system resources from applications in user space to the operating system kernel.

The *Virtual File System* layer is an indirection layer, providing the data structures and interfaces needed for system calls related to a standard Unix file system. It defines a common interface that allows many kinds of specific file systems to coexist, and enables the default processing needed to maintain the internal representation of a file system. The VFS runs in a highly concurrent environment as its interface functions may be invoked by multiple, concurrently

executing application programs. Therefore, mechanisms implementing mutual exclusion are widely used to prevent inconsistencies in VFS data structures, such as atomic values, mutexes, reader-writer semaphores and spinlocks. In addition, several global locks are employed to protect the global lists of data structures while entries are appended or removed. To serve a single system call, typically multiple locks have to be obtained and released in the right order. Failing to do so could drive the VFS into a deadlock or an undefined state.

Each *specific file system*, such as EXT2, EXT3 and ReiserFS, then implements the processing for supporting the file system and operates on the data structures of the VFS layer. Its purpose is to provide an interface between the internal view of the file system and physical media by translating between the VFS data structures and their on-disk representations. Finally, the lowest layer contains *device drivers* which implement access control for physical media.

Data structures. The most relevant data structures are *superblocks*, *dentries* and *inodes*, whose names are used in different contexts outside the VFS; we employ the VFS-related definitions rather than their file-system-specific meanings or their on-disk representations (cf. Fig. 1). The *super_block* data structure describes the abstract properties of the file system, such as its type (e.g., EXT3), the physical device on which it resides, its total size, its mount point and a pointer to the root dentry. The `struct super_block` is defined in `include/linux/fs.h`.

The *dentry* data structures collectively describe the structure of the file system. Each dentry contains a file's name, a link to the dentry's parent, the list of subdirectories and siblings, hard link information, mount information, a link to the relevant super block, and locking structures. It also carries a reference to its corresponding inode, and a reference count that reflects the number of processes currently using the dentry. Dentries are hashed to speed up access; the hashed dentries are referred to as the Directory Entry Cache, or *dcache*, which is frequently consulted when resolving path names embedded within function calls. The `dentry` struct is defined in `include/linux/dcache.h`.

The *inode* data structure carries information specific to a file, whether it is a regular file, directory or device. This includes a link to the relevant super block, backward links to the dentries referencing the inode, file permissions, file type, file size, operations for use on inodes by the VFS, callbacks to the specific file system, device-specific information, and information about how the file is memory-mapped, e.g., it links to file objects which capture the data needed to support file descriptors in user space. The struct `inode` is defined in `include/linux/fs.h`.

Implementation. The public interface of the Linux 2.6.18 VFS consists mainly of the header files `fs.h`, `namei.h` and `dcache.h` residing in `include/linux`. The implementation of system calls can be found in the `fs` subdirectory of the kernel source tree. Here, the files `dcache.c`, `namei.c`, `inode.c`, `stat.c` and `open.c` are notable; they contain the logic for the system calls featured in our model.

To explain the interaction between the different parts of the VFS, we take the `creat()` system call as an example. The functions involved comprise roughly 5k lines of source code, not including data structure definitions and macro expansions. In POSIX, the signature of `creat()` is defined as:

```
int creat(const char *pathname, mode_t mode);
```

providing the full path to the file to be created and the desired file permissions. In the following we discard all permission handling. The VFS entry point for `creat()` is the function `sys_creat()` defined in `open.c` which redirects to

```
sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

Therefore, `creat()` is handled as a special case of the `open()` system call. `sys_open()` then triggers `do_sys_open()` that calls `do_filp_open()`, which in turn invokes `open_namei()`. This last function resides in `namei.c` and represents the main part of the `open` routine. It first uses `do_path_lookup()` to traverse the dentry directory tree. This involves increasing and decreasing usage counters by calling `dget()`/`dput()` from `dcache.c` and obtaining locks for dentries belonging to the path. If at least the parent directory of the file to be created exists, `do_path_lookup()` returns successfully, passing a pointer to the parent's dentry. If the target file for the `creat()` operation does not yet exist, the path lookup function will return a dentry that is not yet associated with an inode. In that case, `open_namei()` invokes `vfs_creat()` to propagate the creation of an inode down to the specific file system and link the newly created inode to the dentry. At this point, the file creation is complete.

Additionally, the process of creating a file involves obtaining and releasing several reader-writer semaphores as well as the `i_mutex` of the parent's inode. It also has to obtain global spinlocks protecting complete lists of dentries and inodes, in case the execution of the system call is preempted by the scheduler.

Properties. One aim of our study is to show the absence of two principal types of error in the VFS implementation. Firstly, since locking mechanisms are a key part of the implementation, we are interested in the existence of scenarios that might cause *deadlock*. Secondly, since some fields of the data structures are logically dependent, we wish to establish *integrity* properties implying that the file system is being maintained in a consistent state. These properties encapsulate relationships between the data structures that ought to hold universally or between calls to VFS functions. These integrity properties are of three types:

(a) *Allocation properties*, expressing that the information pointed to by the fields of assigned nodes is allocated;
(b) *Reference properties*, expressing that reference counters are maintained in a healthy way by the functions operating on the VFS;
(c) *Structural properties*, expressing static relationships between the data structures of the VFS, which ought to be maintained by the functions operating on these structures.

3 Extracting and Validating our VFS Model

This section presents our model of the Linux VFS implementation, discusses our adopted methodology for extracting the model, justifies our key modelling decisions, and summarises our approach to validating the model.

Initial considerations. The Linux VFS implementation is very large in size: approximately 65k lines of C code are directly relevant to the VFS and must be analysed thoroughly, whilst roughly 80k lines of code which include details of, e.g., memory and scheduling, are less directly relevant but require consideration nonetheless. Concurrency mechanisms and the use of macros also add to the complexity. Therefore, automation of the modelling process is a key concern.

To this end, we initially explored Modex [18] which can be used to extract high-level SPIN models directly from C source code. The aim was to use Modex in the initial phase of modelling to produce a slice of the VFS implementation. Unfortunately, despite attempts to simplify the task by, e.g., preprocessing the source code, Modex failed to parse the kernel source. This was likely due to non-standard and compiler-dependent source code fragments. We also considered software model checkers, such as BLAST [15], but found them to be ineffective for similar reasons. In the end, the only recourse was to use automation less directly, to support an essentially manual modelling process.

To facilitate this, we needed to carefully analyse the VFS implementation to identify the data structures that have to be captured by the model, and the integrity properties that these structures ought to satisfy. Despite the wealth of available information on the Linux VFS, these usually comprise English language descriptions of the data structures together with associated operation signatures. In contrast, [11] provides a formal specification of part of the POSIX interface. However, this is too abstract for our purposes as it avoids VFS-level considerations such as the maintenance of internal data structures and how locking mechanisms are employed. Because we required precision in order to produce an accurate model, our model had to be derived from the source code itself.

Modelling decisions. Since our case study is embedded within a research project, the scope of our VFS model had to be adjusted so as to fit the project's schedule. Therefore, we chose to incorporate only the basic operations on objects in the file system: creating files and directories, and deleting files and directories. Other POSIX commands, e.g., regarding mounting and links, were deferred; this means that only a single superblock structure is necessary. In addition, we treat files as atomic entities, thus abstracting from file content.

For practical purposes, it is necessary to impose a limit on the size of the file system that the model is to maintain. To keep state spaces tractable, we set this limit to eight nodes including root. The choice of eight nodes means that we were able to investigate operations on non-trivial configurations of the file system whilst remaining within the bounds imposed by current model-checking technology. In particular, we avoided modelling the dentry hash table as it is an unnecessary cost given the eight-nodes limit; the hash table look-up function can instead be modelled efficiently as a search over all dentries.

A final modelling decision underlying this case study embraces a dynamic file structure where links are explicit, rather than a fixed structure where links are implicit and inferable from each node; this means that we were able to remain more faithful to the implementation.

Representing the VFS data structures. Central to the VFS implementation are the logical structures of the file system (including parents, siblings and subdirectories) and the locking mechanisms (including spinlocks, mutexes and reader-writer semaphores). Consequently, the superblock, dentry and inode data structures of the VFS implementation must be represented in our model, and the most significant issue becomes which of the structures' fields should be included, how to represent them, and which fields can be omitted.

The process of identifying the fields of interest must be based on the information contained in the header files and consider how each field is used by the VFS implementation. Full details of our engagement in this lengthy process can be found in a technical report [13]. For example, for the dentry table, the key fields are: (i) *d_lock*, since locking is essential for concurrency; (ii) *d_inode*, *d_parent*, *d_child* and *d_subdirs*, since these capture the structure of the file system; (iii) *d_count*, which records whether a dentry is assigned and the number of processes accessing the dentry. Additionally, an *is_allocated* boolean flag was introduced to model dynamic data allocation.

The next decision is how to represent each field and estimate the number of bits required. Our parameters for the *dentry* table are: (i) *d_lock* is assigned three bits: one for the status of the lock, one for the process holding the lock (up to two processes), and one indicating a waiting process; (ii) *d_inode* and *d_parent* are assigned three bits each, allowing one to reference a maximum of eight inodes and dentries, whereas *d_child* and *d_subdirs* are allocated eight bits each, allowing up to eight siblings and children of a dentry to be marked rather than stored as a linked list; (iii) *d_count* is allocated three bits, permitting up to two processes to access a dentry at a time, with one bit contingency. In addition, *d_iname* is allocated three bits, allowing for eight different names and giving a maximum directory structure width of seven (plus the root).

Extracting the VFS model. The aim of the extraction process was to isolate the algorithms operating on the identified VFS data structures, and to express these in C syntax in an abstract way. The choice of using C as the modelling language also simplified the validation of our model, since it eases comparisons between model and implementation and since it can easily be fed into simulators.

As indicated earlier, the task of extracting a model from the VFS implementation was made difficult by a number of factors, including the size of the implementation and the heavy use of dynamic memory allocation and function pointers. Concurrency issues also contributed to the complexity of the exercise. To provide at least some automated support for the task, we generated call traces from kernel executions, which allowed us to obtain a series of algorithmic "snapshots" and thus an accurate impression of functionality and ordering. For example, by analysing the traces for `sys_creat()`, it was possible to confirm its behaviour that we presented in Sec. 2.

To obtain traces from a running Linux kernel we adopted the *Kernel Function Trace* tool (KFT, *tree.celinuxforum.org/CelfPubWiki/KernelFunction*) to work with Linux 2.6.18, implemented a few simple test drivers that initialised KFT for a particular system call such as `sys_creat()`, executed the call and

obtained the trace. KFT itself employs the *finstrument-functions* (*gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.htm*) capability of the compiler to add instrumentation call-outs to every function entry and exit, which are used to dump the jump and return addresses to a trace log. With the help of the kernel's symbol table, the log entries can be translated into their respective function names. However, the view of the VFS we obtained from call traces is necessarily incomplete, and a great deal of effort still had to be spent manually inspecting the code. This is due to several reasons: (i) call traces do not reveal how a particular function operates on the VFS data structures of interest; (ii) macros are not instrumented; (iii) several important function calls are missing in each trace since some functions cannot be instrumented; this is because they have to be called from an atomic context in which performing blocking I/O operations, i.e., writing out a log message, is not permitted.

Using call traces and manual inspection we were able to model the core behaviour of the VFS within several person weeks. Table 1 presents the model fragment which we extracted for the `creat()` function discussed above. Similar fragments were produced for the system calls `sys_unlink()`, `sys_mkdir()`, `sys_rmdir()` and `sys_rename()`, for various additional VFS functions such as `path_lookup()` and `path_release()`, as well as for functions that belong to other parts of the kernel's infrastructure. Due to space constraints — the complete model is about 3k lines — we cannot show it in full here. However, the final model can be downloaded from *research.nianet.org/~radu/VFS/*.

Validating the VFS model. In the absence of full automation, we adapted two classic techniques for validation: (a) our final model was extensively *reviewed* and cross-checked against the implementation, with an overall effort of two person months; (b) a similar effort was spent in *simulating* our model.

For conducting the simulation runs, we employed the SPIN verifier [17] for two reasons. Firstly, the syntax of SPIN's input language Promela is close to the C syntax adopted by our model. Therefore, the translation could be performed quickly and with little risk of introducing errors. Secondly, SPIN's rich simulation capabilities, along with the ability to add assertions, allowed for a rigorous testing regime to be implemented. To aid simulation, our SPIN model was confined to a single process, thereby eliminating the complexity introduced by concurrency. More than 100 different simulation runs were conducted on the SPIN model that was heavily injected with assertions; about 3% of the model's lines are assertion statements. Each run was performed as a simulation of one of the system calls from a recognised 'healthy' state, involving creating/deleting existing/non-existing files/directories at various levels, attempting to delete the root and copying a directory onto itself, etc. Several early errors in our VFS model were identified and corrected by these means.

Part of the model validation was also carried out during the verification phase, which involved the model checker SMART [6] as described below. Indeed, several errors were eliminated as part of the SMART modelling process where, in the first instance, model discrepancies such as unexpected verification results, property violations and deadlocks were treated as potential signs of an invalid

Table 1. Model fragment for sys_creat().

```
int sys_creat (string path) {
 lookup_res_t l;
 inode_t itmp;
 dentry_t parent, file;

 l = path_lookup (path);
 parent = *l.parent;
 file = *l.file;
 if (!parent.is_allocated)
  {
   if (file.is_allocated)
    /* deals with root look up */
    { dput(file); }
   return (ERROR);
   }

 down (parent.d_inode->i_mutex);

 if (file.is_allocated &&
      !is_directory (file))
  { up (parent.d_inode->i_mutex);
    path_release (file);
    return (SUCCESS); }

 if (file.is_allocated &&
    is_directory (file))
  { up (parent.d_inode->i_mutex);
    path_release (file);
    return (ERROR); }

 spin_lock (dcache_lock);
 file = allocate_dentry(
   last_component(path), parent);
 if (!file.is_allocated)
  { spin_unlock (dcache_lock);
    up (parent.d_inode->i_mutex);
    dput (parent);
    return (ERROR); }
 dget (file);

 spin_lock (inode_lock);
 itmp = allocate_inode(file);
 file.d_inode = &itmp;
 spin_unlock (inode_lock);
 if (!file.d_inode->is_allocated)
  { atomic_write (file.d_count, 0);
    dput (parent);
    spin_unlock (dcache_lock);
    up (parent.d_inode->i_mutex);
    return (ERROR); }

 update_parent(
    *((dentry_t *)file.d_parent));
 path_release (file);
 spin_unlock (dcache_lock);

 up (parent.d_inode->i_mutex);

 return (SUCCESS);
}
```

abstraction. For each discrepancy, the VFS model was re-checked against the VFS implementation and, if appropriate, revised.

4 Verifying our VFS Model Using SMART

The next step in our case study was to verify our validated VFS model for absence of deadlock and adherence to data-integrity constraints using model-checking technology. To do so, we initially attempted to run SPIN in verification mode on the (single-process) SPIN model and established freedom from assertion violations. However, the analysis on a modern PC failed for all but the most trivial configurations that involve two or three nodes only, since the sizes of the state vector and the reachable state spaces are simply too large to be represented explicitly, even using advanced features such as collapse compression.

Also model checkers such as BLAST [15] cannot deal with our VFS model, due to the presence of concurrency in the VFS environment.

We therefore shifted our focus to SMART, which is a state-of-the-art symbolic model-checker for concurrent systems [6] with which we are most familiar. SMART implements the *Saturation* algorithm [8] which exploits properties of interleaving semantics for manipulating decision diagrams and can be significantly more time- and memory-efficient than SPIN [7]. As SMART provides a notation based on Petri nets rather than a software modelling language, we had to rephrase our VFS model into a Petri net. This involved introducing a program counter and circumventing the unavailability of advanced (and recursive) data structures. In addition, our SMART model had to comply with a restriction imposed by Saturation which, informally, demands that Petri net places that are functionally dependent on others be grouped in the same net partition [8].

Our VFS model as a Petri net. As for the VFS model in C, the SMART model is parameterised by the maximum number of dentries (ND), inodes (NI), and concurrent processes (NP). The encoding employed for translating the VFS model to SMART is rather straightforward: variables are represented as Petri net places, and instructions are represented as Petri net transitions. Moreover, the fields of the dentry and inode data structures are represented as arrays, i.e., the *d_parent* field of dentry k is the element $d\_parent[k]$.

In constructing the SMART model, the VFS algorithms and data structures were abstracted in a number of minor ways to make it possible to capture the required behaviour without introducing incidental complexity. One example is the need to represent path arguments to system calls and the traversal of the dentry tree structure. Lists, as used in VFS, are not native to the SMART language, and introducing them artificially would have incurred unacceptable overheads. Instead, our SMART model indexes the fully qualified filenames present in the system with natural numbers. This means that the *d_iname* field could be dropped, which also simplifies the `path_lookup()` function.

Another aspect involves the deallocation of unused nodes, which in the VFS implementation is performed separately by a garbage collection process. Our SMART model assumes an "as early as possible" deallocation in order to sequentialise deallocation and minimise complexity. Further abstractions concern the `d_subdirs` field that is used to record whether the dentry is a directory, rather than the identity of its sub-directories, and the `d_child` field that is used to record the number of siblings, rather than the identity of the siblings.

Again, the resulting VFS model cannot be reproduced here due to space constraints — e.g., the SMART code for the `creat()` function alone is 650 lines —, but is available for download from *research.nianet.org/~radu/VFS/*. The VFS model ranks with the most complex systems ever modelled in SMART. This perspective is not only reflected by the sheer size of the model (2,900 lines of SMART code), but also by the inherent complexity of the VFS. For comparison, two other similar industrial-size applications modelled in SMART are *NASA's Runway Safety Monitor* [22] (1,850 lines) and *NASA's clock synchronisation and self-stabilisation protocols* [20] for the SPIDER architecture (1,190 lines).

Integrity properties. As stated in Sec. 2, we wished to verify that the VFS model is free of deadlock and that it maintains integrity properties on its data structures. For the latter, we concentrated on the following three properties:

(a) If a node is assigned, then its parent is allocated. Here, 'assigned' means that the node itself is allocated and marked as in use (i.e., $d\_count > 0$). This is an allocation property that may be formalised by:

$$\forall d1, d2 : Dentry \bullet\ d1.is\_allocated{>}0 \wedge d1.d\_count{>}0 \wedge d1.d\_parent{=}d2 \Rightarrow d2.is\_allocated{>}0.$$

(b) When the system is stable, i.e., between file system operations, all allocated nodes' d_counts are either 0 or 1. This means that a node's reference counter does not imply that the node is in the process of inspection or alteration between operations. This is a reference property that may be formalised by:

$$\forall d : Dentry \bullet\ stable \wedge d.is\_allocated{>}0 \Rightarrow d.d\_count{=}0 \vee d.d\_count{=}1,$$

where predicate *stable* is defined using the value of the program counter.

(c) The only cycle in the parent relation is the one on root. (By default, the parent of the root is itself.) This is a structural property that, for a file system of at most eight nodes, may be formalised by:

$$\forall d1, d2, d3, d4, d5, d6, d7, d8 : Dentry \bullet$$
$$d1.is\_allocated{>}0 \wedge d1.d\_count{>}0 \wedge \ldots \wedge d8.is\_allocated{>}0 \wedge d8.d\_count{>}0 \wedge$$
$$d1.d\_parent{=}d2 \wedge \ldots \wedge d7.d\_parent{=}d8 \Rightarrow d8{=}root.$$

Formulating the required properties in SMART, including deadlock freedom, amounts to re-expressing them as simple operations over decision diagrams. This is straightforward except for the cycle-freedom property which we capture as a set of properties: "no cycles of length one, except for root"; "no cycles of length two"; "no cycles of length three", etc.

Table 2. State-space generation results for SMART.

ND	NI	states	time (sec)	mem. (MB)	ND	NI	states	time (sec)	mem. (MB)
1 process					2 processes				
2	2	325	1.02	1	2	2	222,715	258.49	223
3	3	12,077	9.94	12	2	3	222,715	318.77	233
4	4	1,085,247	77.27	131	3	2	-	-	>8,000
5	5	173,247,829	1,056.88	2,147	3	3	-	-	>8,000

Verification results. Before verifying the properties of interest, it was necessary to construct the state space of the model. Various instantiations of ND, NI and NP were examined. Table 2 lists the results when conducting our experiments using the 64bit version of SMART on a 3.2GHz machine with 8GB of memory running Redhat Enterprise Linux version 2.6.9-5ELsmp. The most significant

contributor to the complexity is the number of concurrent processes, with NP≥3 unanalysable, and NP=2 with ND>2 also exceeding memory.[3]

The integrity properties formalised above were checked successfully against the generated state space, for each instantiation of ND, NI, and NP. Additionally, we verified the following properties: "root is always allocated"; "root is always in use", i.e., $d\_count>0$ is an invariant; and "the parent of an assigned node is in use". Each property was checked for each configuration in negligible time of less than one second, and shown to hold. Collectively the properties imply that every node currently in use is connected to the root.

We also checked each configuration for deadlocks. Initially, the model contained deadlocked states, for the truly concurrent setting (NP=2). Further analysis revealed that this was due to the implementation's critical use of a structure that had been abstracted away, the *dcache*. An extra bit was added to the *Dentry* structure to represent the missing information, and the model was revised accordingly. The model was then shown to be deadlock free, taking negligible time of less than one second for each configuration.

A livelock scenario was also uncovered by SMART when two processes attempt to `unlink()` the same file simultaneously. After an extensive analysis of the source code, this was attributed to the abstraction of protocols and scheduling policy designed to ensure fairness over the way spinlocks were accessed. Unfortunately, little documentation exists on the actual implementation of the scheduler for us to be able to give a firm verdict on whether the scenario is a false positive. However, this shows that modern model checkers can check more than safety properties on our VFS model. Indeed, from a model-checker's standpoint, we investigated three categories of properties: (i) safety properties which, once the reachable state space is constructed, require a single decision-diagram operation; (ii) deadlock which requires a single backward image computation on the reachable state space; (iii) livelock which requires a fixed-point computation.

5 Related Work

While [9, 10] provide techniques for verifying the correct use of file system interfaces represented as finite-state machines, work on verifying properties of file system implementations is relatively scarce.

An ongoing research project on verifying a POSIX-compliant file store, from the application interface down to the data representation on a physical medium, is outlined in [12]. Current work focuses on the construction of formal models of NAND flash memory and the commands that are used to operate it.

A correctness proof for a basic file system with standard data structures and fixed-sized disk blocks is presented in [2]. It uses the Athena theorem prover and employs the constructive approach to verification (as does [11]). The Athena

[3] The state spaces for two processes and ND=2 are indeed identical for NI=2 and NI=3. This is because the third inode is never used in this configuration, since ND<NI and because the allocation policy always returns the first available index.

model involves some of the data structures covered here and also their respective media representation. The work differs from ours in that it does not deal with concurrency-related issues and does not consider a 'real' file system implementation, thereby avoiding the process of model extraction.

Actual file system implementations are studied by Engler et al. in [23, 24]. In [24], model checking is used within the systematic testing of EXT3, JFS and ReiserFS. The verification system consists of an explicit-state model checker running the Linux kernel, a file system test driver, a permutation checker which verifies that a file system can always recover, and a recovery checker using the *fsck* recovery tool. The system starts with an empty file system and recursively generates successive states by executing system calls affecting the file system. After each step, the system is interrupted and *fsck* is used to check whether the file system can recover to a valid state. This approach is combined in [23] with symbolic execution for generating pathological test cases. In contrast to our work, [23, 24] employ run-time verification techniques that cannot exhaustively explore the implementation's state space. However, an advantage over our work is that these techniques do not require manual model extraction.

Verification approaches that model-check the source code of operating system components are presented in [3, 5, 15]. In theory, these are able to prove a file system implementation to be, e.g., free of deadlock. However, as shown in [21], the model checkers employed in [3, 5, 15] also require manual preprocessing of source code. An approach to verifying the implementation of a microkernel's paging mechanism, including a hard disk driver implemented in a fully formalised subset of C and inline assembly, is presented in [1].

6 Conclusions and Future Work

In response to Joshi and Holzmann's mini challenge, we have constructed and verified a small model of several key components of the Linux Virtual File System (VFS). This proved to be a challenging task since current automated techniques for extracting models from C source code cannot deal with important aspects of operating systems code, including macros, dynamic memory allocation, architecture- and compiler-specific code, and inlined assembly. Extracting our model by hand was made especially difficult and time-consuming by the VFS implementation's concurrency mechanisms, uncommon coding styles, and the sheer volume of code. Much time was spent in validating our model via reviews and simulation runs in SPIN. Using the SMART model checker, this model was then shown to respect data-integrity properties and to be deadlock free. The three variants of our VFS model, in C syntax, SPIN's Promela language and SMART's Petri nets, are available for download from *research.nianet.org/~radu/VFS/*.

Our case study clearly demonstrates the feasibility of abstracting data structures and algorithms from a complex file-system implementation, analysing their behaviour via simulation and model checking, and inferring conclusions about the implementation's correctness. However, automated extraction of faithful models is paramount in analytical software verification and must be a continuing

focus for research. This must involve not just program slicing but also representational changes in data structures and algorithms.

Some take-away messages. Here is what this VFS case study has taught us personally, in general terms and regarding various aspects of our work:

Goal: It makes a big difference whether one targets "bug discovery" (debugging) or "bug absence" (verification).

Automation: There is a stringent need for automating model extraction, but no existing tool is mature enough to have served our purpose.

Soundness: Building multiple models is important for fully understanding the underlying system; however, our 'staged' approach could be strengthened by checking the links between the stages formally.

Complexity: Certain aspects of the system, such as the operating system's scheduler which is external to the VFS code, cannot be faithfully modeled without dramatically increasing the size and complexity of the model.

Scalability: The fact that modern model checkers cannot handle larger parameters of our model should not be seen as a deterant since model checking technology is improving quickly.

Future work. It would be valuable to extend the scope of our case study. For instance, considering more functionality such as the specific file system layer would enable more direct comparisons with other approaches to the mini-challenge, e.g., [12]. An alternative way to extend the scope would be to incorporate an abstract model of the scheduler which, e.g., would allow one to adequately check for the absence of livelocks.

Acknowledgments. We thank the reviewers for their insightful comments, and in particular for suggesting the inclusion of 'take-away' messages.

References

[1] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *TACAS*, volume 4963 of *LNCS*, pages 109–123, 2008.

[2] K. Arkoudas, K. Zee, V. Kuncak, and M. C. Rinard. Verifying a file system implementation. In *ICFEM*, volume 3308 of *LNCS*, pages 373–390, 2004.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, volume 2057 of *LNCS*, pages 103–122, 2001.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.

[5] S. Chaki, E. M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2-3):129–166, 2004.

[6] G. Ciardo, R. L. Jones III, A. S. Miner, and R. Siminiceanu. Logic and stochastic modeling with SMART. *Performance Evaluation*, 63(6):578–608, 2006.

[7] G. Ciardo, G. Lüttgen, and A.S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *FMSD*, 31(1):63–100, 2007.

[8] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS*, volume 2031 of *LNCS*, pages 328–342, 2001.

[9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68. ACM, 2002.
[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69. ACM, 2001.
[11] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/EVES: An experiment in the verified software repository. In *ICECCS*, pages 3–14. IEEE, 2007.
[12] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. In *ICECCS*, pages 153–162. IEEE, 2008.
[13] A. Galloway, J. T. Mühlberg, R. Siminiceanu, and G. Lütgen. Model-checking part of a Linux file system. Technical Report YCS-2007-423, U. of York, UK, 2007. Available at *www.cs.york.ac.uk/ftpdir/reports/YCS-2007-423.pdf*.
[14] The Open Group. The POSIX 1003.1, 2003 Edition Specification.
[15] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, volume 2404 of *LNCS*, pages 526–538, 2002.
[16] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
[17] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
[18] G. J. Holzmann and M. H. Smith. Software model checking – Extracting verification models from source code. In *FMPEDS*, pages 481–497. Kluwer, 1999.
[19] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.
[20] M. R. Malekpour. A Byzantine fault-tolerant self-stabilizing protocol for distributed clock synchronization systems. Technical Report TM-2006-214322, NASA Langley Research Center, 2007.
[21] J. T. Mühlberg and G. Lüttgen. Blasting Linux code. In *FMICS*, volume 4346 of *LNCS*, pages 211–226, 2006.
[22] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. *STTT*, 9(1):63–76, 2007.
[23] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *Security and Privacy*, pages 243–257. IEEE, 2006.
[24] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pages 273–288. USENIX, 2004.