Model-checking Part of a Linux File System[*]

Andy Galloway[†], Jan Tobias Mühlberg[†], Radu Siminiceanu[‡] and Gerald Lüttgen[†]

[†]Department of Computer Science, University of York, UK
[‡]National Institute of Aerospace (NIA), Hampton VA, USA

Abstract

We present our experiences with model checking part of a Linux file system. The 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 primary aim of the work was to construct a larger scale case study upon which to measure our own research into model-checking technology. However, the choice of case study material was influenced by the aforementioned mini-challenge. The secondary aim of the work was to add to the existing confidence in the Linux code, for example by "testing" improbable situations through exhaustive model checking. Two models were produced: a Promela model, which was analysed using SPIN, and a Petri-Net-based model, which was analysed using versions of the SMART model-checker. The approach adopted was incremental, initially focussing on the basic functions of Linux's Virtual File Systems layer. The report presents intermediate results.

[*] We gratefully acknowledge the support of the UK EPSRC, who funded the research under grant GR/S86211/01.

Contents

1. Introduction

In [Hoa03], Hoare proposes a 15-year grand challenge which calls on the program verification community to collaborate on building verifiable programs. At the first Verified Software: Theories, Tools, Experiments conference [Eth05,MW07], Joshi and Holzmann [JH05] proposed a more modest *mini-challenge*, as a significant stepping stone towards meeting Hoare's challenge. Their mini-challenge was to build a verifiable file system, such as a file system conforming to the POSIX interface standard [Ope03].

Program verification in the context of Hoare's challenge specifically involves *full demonstration* of program correctness, rather than more general uses of the term "verification" (e.g. testing). Broadly speaking, there are two approaches to program verification. On one hand there is the *constructive* approach, in which formal reasoning is first employed to establish the validity of a specification and then the correctness of an implementation with respect to such a specification. On the other hand there is *analytical* approach, which aims to build a valid abstract model of an existing implementation and show that this model satisfies some set of correctness criteria. The former approach is emphasised by methods such as Z [Spi95], B [Abr96], VDM [Jon90], whilst the latter is the focus of certain "model-checking" approaches such as SPIN [Hol03].

In this report we present our experiences with a file system case study, in the spirit of Joshi and Holzmann's mini-challenge, in which we applied an analytical approach to verification. The case study involved producing an abstract model of part of the Linux kernel and analysing the model using two distinct model-checkers, SPIN and SMART. The report describes the intermediate findings of the project.

The remainder of the report is organised as follows: section 2 describes the aim of the project and section 3 introduces the Linux file system architecture. In section 4 we describe the scope and methodology used in the study. Sections 5 and 6 concentrate on abstraction; section 5 explains how the variables and data structures of the Linux file system were abstracted, and section 6 focuses on abstraction of the Linux code (i.e. algorithms). Sections 7 and 8 discuss the models produced by the study – one in Promela/SPIN and one in SMART. Section 9 presents related work, and section 10 presents the conclusions of the study. There are five appendices, which cover an example of part of a Linux header file, the information abstraction process, the pseudocode produced by abstracting the Linux code, extracts from the SPIN model and extracts from the SMART model.

2. Aims

The work described formed part of a larger body of work into efficient symbolic model-checking for interleaving systems [CLS01,CJMS06,CLM07,ELC07,ELS07, YCL07a,YCL07b]. In particular, collaborative work between the University of California, US, Iowa State University, US, the NIA (National Institute of Aerospace), US and the University of York, UK, has resulted in a tool, SMART [CJMS06], which implements efficient model-checking algorithms for interleaving systems. The tool forms important background for the study.

The work had two complimentary aims:

1. To produce a large case study on which the performance of the SMART model-checker (and prototypes for variants implementing parallelised algorithms) could be measured and compared.
2. To assess the feasibility of analytical program verification on part of the Linux kernel.

The first aim was primary. Existing case studies for SMART tended to be modest sized academic examples, and we were searching for a larger example, of real-world significance, to provide a further benchmark.

The secondary aim attempted to add to existing confidence in its correctness of the Linux kernel. In particular, the goal was to assess the effectiveness of model-checking technology for this purpose – for example, by analysing scenarios that were unlikely to arise in testing or in subsequent use. Three potential findings were envisaged, with increasing likelihood: i) the corruption (deviation from intent) of the underlying data state, deadlock or livelock, leading to an observable error (bug); ii) the (possibly transient) corruption of the underlying data state, not leading to an observable error (but at risk of revealing itself in future revisions); iii) an absence of errors, hopefully contributing to evidence of the correctness of the implementation.

3. The Linux File System Architecture

The logical architecture of a Linux file system is shown in Figure 1. The elements shown are as follows [Bov02]:

- Application: The user program.
- C POSIX library: Provides functions facilitating file access as defined by the POSIX Standard [Ope03], e.g. open file *open()*, delete file *remove()*, make directory *mkdir()*, and remove directory *rmdir()*.
- System call interface: Propagates requests for system resources from the application program into the operating system kernel.
- Virtual File System (VFS): This layer is an *indirection* layer. It provides the data structures and interfaces needed for system calls related to a standard Unix file system. Its main strength is to provide a common abstract interface allowing many different kinds of specific file systems to coexist. The VFS data structures are instantiated when a file system is mounted with the appropriate call addresses into the Specific File System layer. The VFS also provides some default processing and caching for its data structures. Since the VFS is used in a multi-threaded environment, serving system calls from within different process contexts, it incorporates various locking mechanisms in order to sequentialise concurrent access to file systems.
- Specific File System (e.g. EXT2, EXT3, FAT): This layer provides the processing supporting a particular file system. The specific file system operates on the data structures provided by the VFS. Its purpose is to provide an interface between the VFS and the physical storage by transforming the VFS data structures into their on-disk representation and back. Therefore it defines the data structures used by the media representation, and manages the way elements of the file system are read in from, and written out to, the media.
- Device Drivers: These implement access control – i.e. reading from and writing to – the physical media.

3.1 Concurrency in the VFS

The VFS runs in a highly concurrent environment in which its interface functions can be invoked from multiple concurrently executing application programs. Hence, computer architectures supporting symmetric multi-processing, as well as normal process preemption caused by scheduling on single processor machines, gives rise to the indeterminate sequencing of the respective threads. Therefore, mechanisms implementing mutual exclusion are widely used in order to prevent inconsistencies arising in this context. In the case of the VFS that means that each internal data structure consists of multiple mutual exclusive components such as *atomic values*, *mutexes*, *reader/writer semaphores* and *spinlocks* (cf. [CRK05]) assuring sequentialisation of operations manipulating these structures. In addition to this, several global locks are employed to protect the global lists of data structures while entries are appended or removed. In order to serve a single system call, usually multiple locks have to be obtained and released in the right order. Failing to do so can drive the VFS into deadlock conditions or undefined states due to memory corruption.

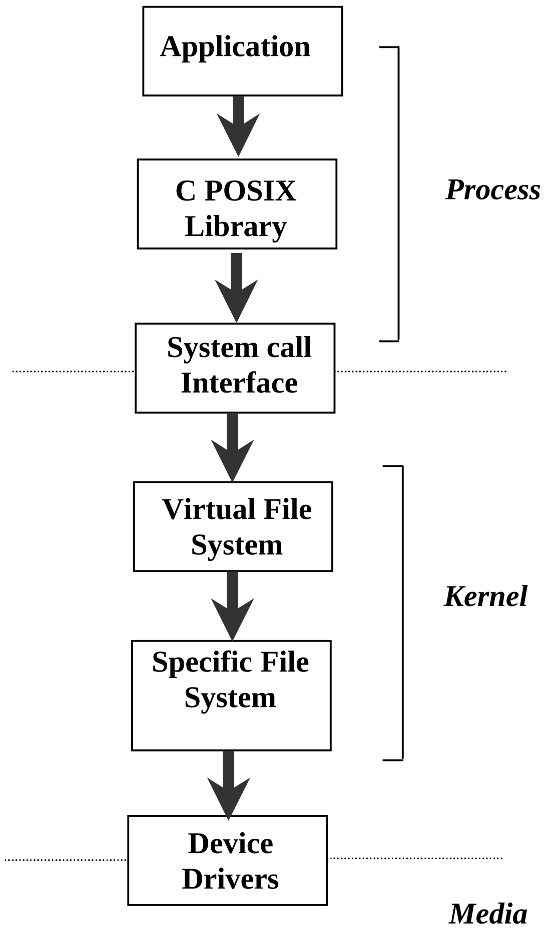

Figure 1. The logical hierarchy of the Linux file system.

3.2 Data Structures in the VFS

Of the information structures which make up the file system, the most important are *super blocks*, *dentries*, and *inodes*. In most cases the names of these structures are used in different contexts outside the VFS. In the following we only consider the VFS-related structure definitions and not their file system specific components or their on-disk representations. Further details on these structures can be found in [Bov02] and in the respective parts of the kernel header files.

- *super_block* objects describe the abstract properties of the file system, such as its type (e.g. EXT2), the physical device on which it resides, the total size of the file system and the mount point. They also point to the root *dentry*, as well as lists of all further dentries and inodes of the file system, and contain several structures used for locking. Furthermore, there are several flags defining the characteristics of the file system (e.g. whether it is read-only). Operations for manipulating the super block by the VFS and from the system call interface as well as callback functions for the underlying specific file system are stored as function pointers. The super blocks of all currently mounted file systems are stored in a circular doubly linked list. At the device level, similar super block structures are maintained on the media – many specific file systems (e.g. EXT2) maintain multiple instances of their super block for fault tolerance (and recovery) purposes. The struct *super_block* is defined in *include/linux/fs.h* in the Linux source hierarchy.

- *dentry* objects collectively describe the structure of the file system. The dentry contains the file's name, though the file need not be a regular file – for example it can be a directory or device. There is also a shortened version of the name, which is used in hashing the dentries for speed of access (the Directory Entry Cache, *dcache*). Other important fields include: the parent of the dentry (root points to itself), the dentry's list of children and siblings, operations for use on dentries by VFS and from system calls, hard link information (pointers to other dentries), mount information, a link to the relevant super block, and locking structures. The dentry also carries a reference to its corresponding *inode,* as well as a reference count, which approximately corresponds to the number of processes currently using the dentry. Dentries do not to have corresponding structures at the device level, they merely function as a cache for the information being carried by the corresponding inode structure. The definition of the *dentry* struct can be found in *include/linux/dcache.h*.

- *inode* objects carry information specific to a file (regular file, directory or device). This includes, for instance, a backward link to the dentries referencing the inode, file permissions, file type, file size, operations for use on inodes by the VFS as well as call backs to the underlying specific file system, device specific information (for devices) and information about how the file is memory mapped. At the device level, some specific file systems (e.g. EXT2) use a structure with the name *inode*. However, the information carried by the inode is different at the device level (e.g. subsuming dentry information, sector mapping rather than memory mapping etc.). The struct *inode* is defined in *include/linux/fs.h* in the Linux source.

3.3 Concurrency and Data Structures

As well as the static view of the data structures used by the VFS, there is an additional dimension introduced by concurrency. For example, Figure 2 illustrates how three different processes interact with the same file.

The diagram shows three processes using their own *file* object, two processes are using the same hard link to the corresponding *inode* object. Hence, only two *dentry* objects are required, one for each hard link, pointing to the same *inode*. The components of this picture which are considered as part of the study (see section 4) are highlighted.

3.4 VFS Code Architecture

The VFS implementation as provided by Linux 2.6.18 consists mainly of the three public header files *fs.h*, *namei.h* and *dcache.h* residing in *include/linux* of the kernel's source hierarchy. These files define the public interface including the aforementioned data structures. For example, appendix A provides the definition of the *dentry* struct (from *dcache.h*). Note, however, that most definitions are dependent on other parts of the Linux kernel and employ various external data types and functions.

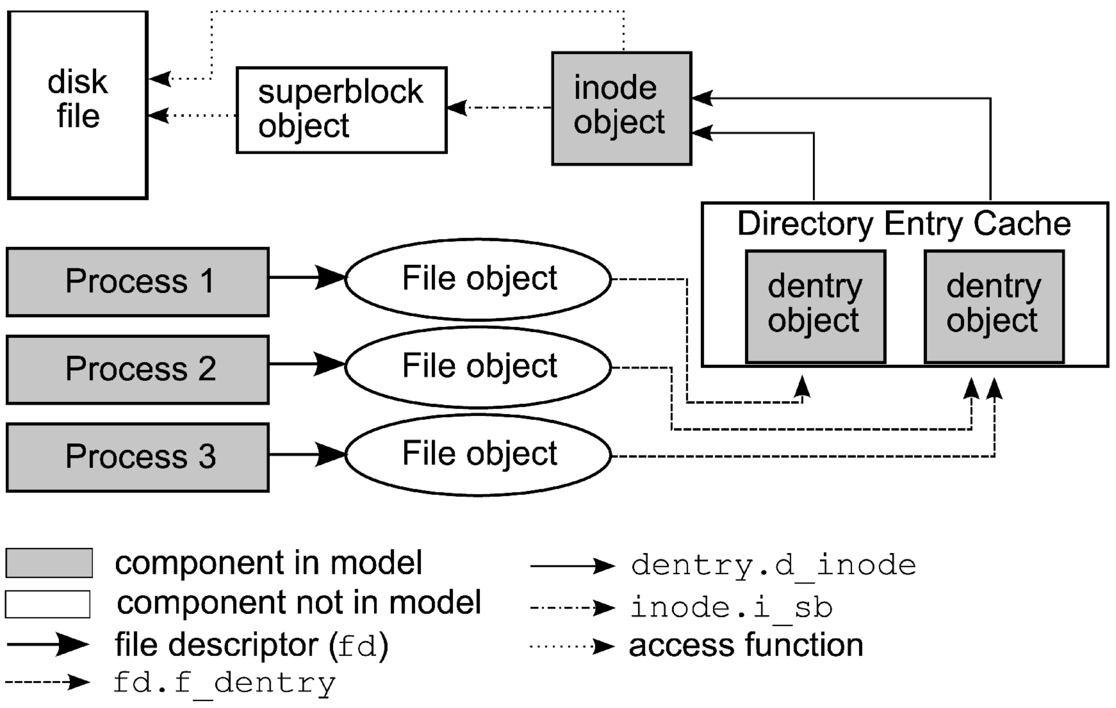

Figure 2. Data Structures Process Perspective

The actual implementations of system calls provided by the VFS can be found in the *fs* subdirectory of the kernel source tree. Given our scope, the files *dcache.c*, *namei.c*, *inode.c*, *stat.c* and *open.c* are of most importance since they contain the logic for the system calls we are most interested in. The VFS implements system calls such as *creat()*, *open()* and *stat()*. Most of these calls have a path name argument passed to them from the calling application program. Resolving these path names and returning the respective dentry is the central part of the VFS's Directory Entry Cache *dcache*.

Let us explain the interaction between the different parts of the VFS on the example of the *creat()* system call. According to POSIX, the signature of *creat()* is defined as:

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

where *pathname* is the full path to the file which is supposed to be created, and *mode* the permissions with which the file should be created. In the following we discard all permission handling.

The VFS entry point from *creat()* is the function *sys_creat()* defined in *open.c*, which redirects the system call to

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

Hence, *creat()* is handled as a special case of the *open()* system call. *sys_open()* then calls *do_sys_open()*, which calls *do_filp_open()*, which finally invokes *open_namei()*. The function *open_namei()* resides in *namei.c* and represents the main part of the open routine. It first uses *do_path_lookup()* to traverse the dentry-representation of the directory tree, starting either from the root directory or from the current working directory of the calling process. This involves increasing and decreasing usage counters (*dget()*/*dput()*, from *dcache.c*) and obtaining locks for dentries belonging to the path. Furthermore, dentries for path components that are not already cached need to be allocated and filled with data obtained from the specific file system implementation using *real_lookup()*. If at least the parent directory of the file to be created exists, *do_path_lookup()* will return successfully, passing a pointer to the parent's dentry. If the target file for the *creat()* operation does not exist yet, the path lookup functions will return with a “negative” dentry, i.e. a dentry, which is not associated with an inode yet. In that case, *open_namei()* will invoke *vfs_creat()* in order 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. Please note that we outline only one path through the open routine, discarding security checks and error cases for simplicity.

Besides the details given above, the process of creating a file involves obtaining and releasing several reader/writer semaphores as well as the *i_mutex* from the parent's inode and the global spinlocks protecting the global lists of dentries and inodes in case the execution of the system call is preempted by the scheduler. We will give more details on this in Section 6**.**

The source code of the functions involved in creating a file on the VFS level comprises of roughly 5k lines of code, not including data structure definitions and macro-expansion. The entire VFS implementation is about 70k lines long.

4. Methodology

The approach adopted was incremental, with the aim of starting with a small abstract model, and then expanding and adding detail. The initial scope, and incremental direction, was defined in several dimensions:

- We decided to concentrate initially on the generic aspects of the Linux file system implementation – the Virtual File System systems layer. The aim was to eventually evolve the model by adding further details supporting a specific file system (and appropriate media), such as EXT2.

- We initially ruled out multi-host concurrency and decided to model just a limited number of (single host) processes. In the case of one of the models (the SPIN model) the decision was taken to start with a single process. The aim was then to extend this by adding further concurrency i.e. additional processes, and possibly multiple hosts.

- As a starting point for the modelling activity, we chose to incorporate only the basic operations on files and directories: create file, remove file, make directory, remove directory and rename. The intention was to extend this in the future to include for example mounting, unmounting, links etc. and other POSIX commands.

- Our approach was to abstract the internal information structures used in maintaining the file system by selecting certain fields. The intention was to expand the selected fields later to include further detail. The initial fields of interest were selected according to their relevance in the scope outlined above – i.e. relevance to VFS layer, concurrency and basic file system operations.

- For practical purposes it was necessary to impose a limit on the size of the file system the models were able to maintain. In order to keep state spaces tractable this limit was set initially to 8 nodes (including root), with the intention to expand later if viable. The limit of 8 nodes was chosen to give each node (e.g. dentry, inode) a 3 bit address, and this was a hard limit imposed by the types of the variables and structures used in the models. We also aimed to be generic with respect to a soft limit of less than 8 nodes, so that models could be executed on various sized file systems up to the hard limit[1].

It had always been the intention to construct a SMART model of the file system implementation; we also took the decision to develop a SPIN model. There were two reasons for this: firstly, the SPIN model would provide a valuable basis for comparison. Comparisons could be made between models, modelling processes and the effectiveness of the model-checking support tools. Secondly, at the outset of the study we believed that the SPIN model would provide a vital intermediate representation of requirements, from which the SMART model could be developed. This turned out not to be the case. A pseudocode (incomplete C) representation was developed to support both modelling activities, and this turned out to be an adequate basis for both modelling activities to proceed more or less in parallel. SPIN was

[1] The soft limit was essentially used to manage state space sizes.

chosen, as opposed to modelling languages, because of its maturity, exposure, and ability to support both compound data structures and concurrency.

As well as the existing SMART model-checker [CJMS06], recent research has produced prototypes which parallelise the algorithms for use in a shared-memory (multi-core) setting [ELC07,ELS07]. Thus, in addition to comparing results from the different models, one of the purposes of the study was to compare results for different variants of SMART.

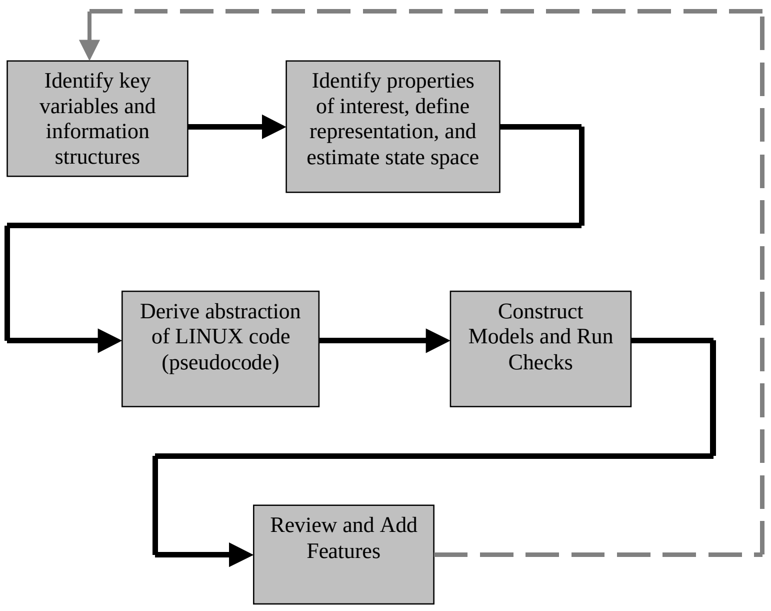
Identify key variables and information structures
Identify properties of interest, define representation, and estimate state space
Derive abstraction of LINUX code (pseudocode)
Construct Models and Run Checks
Review and Add Features

Figure 3. The process followed by the study.

The reason for building the models, other than to draw comparisons, was to attempt to find (potential) errors. The search focussed on two principal kinds of error. Firstly, as part of the process of identifying the information fields of interest we sought to identify consistency properties. These were relationships between the data that should hold whilst the file system is in a *stable* state – i.e. when it is not in the process of being altered in some way. An example of such a consistency property might be for instance that the sibling and child fields for a particular dentry do not contradict the parent fields in its peer dentries. Secondly, locking systems were a key part of the implementation. They existed to ensure mutual exclusion over access to parts of the file system structure (e.g. particular dentries), and thus ensure structural consistency. One of the things we were interested in was whether any situations might exist (however improbable) that might cause the locking mechanisms to jam – i.e. result in deadlock or livelock. An example of such a situation might for instance involve two interdependent processes competing for a spinlock, where their interdependency prevents the spinlock from ever being released (and thus progress being made). From a methodological point of view the intention was that both models would investigate both potential sources of error.

An outline of the process intended to be followed by the study is shown in figure 3. Note that the outline does not show minor iterations. For example, the information structures identified in the first task initially yielded state spaces that were higher than desired. A minor iteration could have been shown around these two activities.

5. Identifying Key Variables and Structures

The variables and structures were selected for inclusion in the models on the basis of the scope defined by the methodology. Initially, this involved information relevant to the VFS level (internal representation), the logical structure of the file system (parents, siblings, children etc.), and the locking mechanisms (spinlocks, mutexes, reader/writer semaphores). Consequently, the superblock, dentry and inode structures were all identified for inclusion, and the most significant issue became which fields from the structures to include, which to omit, and how to represent/abstract the fields of interest. Certain other structures were considered initially, such as the *file* structure (used to maintain the status of files associated with each inode). However, it was decided that concentrating on the file system structure itself was more important initially than modelling access to its contents. Structures such as this were therefore deferred until future iterations.

The process of identifying the structures and fields of interest was based on the information contained in the header files from the kernel source, i.e. the struct definitions for superblocks, dentries and inodes. For each, we considered how each field was used by the file system implementation (this involved considerable investigation in cases were the usage was unclear or subtle). Each field was then assigned a relevance (high, medium or low) according to the scoping rules outlined in the methodology.

Another important consideration was the allocation and deallocation of memory to store the data structures of interest. For the superblock this was not a problem, since there would be just one. However, for dentries and inodes the number of data structures in use at any particular time would vary; new structures would need to be allocated and, potentially, old structures would need to be deallocated. Clearly, in order for any model to be analysable (i.e. have a sufficiently small search space) we would only have a limited supply of variables with which to work. There were two options here, and resolving the choice was essential in deciding which fields to include and which to omit. The first choice was to define a static structure, representing a maximal file system, bounded in width and depth. In this approach potential parent and child etc. links would be implicitly inferable from the file system node in question, and assignment/deassignment could be achieved by marking whether nodes were in use. The second option was to dynamically allocate from a pool of available structures, and maintain the parent/child etc. links explicitly. i.e. produce an abstract model of dynamic memory allocation.

The situation was made more complex by the way that the data structures are maintained in the file system implementation. In the implementation a data structure can have three different kinds of status. It can be allocated and assigned (in use, part of the file system), allocated but deassigned (not part of the file system structure but not deallocated), and deallocated. This is so that reuse and deallocation of unused nodes can occur efficiently[2].

We decided on a strategy based on abstract dynamic memory. The rationale behind this was twofold: firstly it allowed for greater flexibility in file system structures we would be able to model check; secondly, it would be a more faithful model of the implementation both in terms of the way structures are allocated and assigned, and in the way the overall file system structure was maintained (i.e. with explicit links).

[2] i.e. by a separate low-priority clean-up process.

A final consideration was whether (and how) to model the dcache. The dcache involves a hash table allowing efficient access to dentries based on (part of) their name. The issue was whether, strategically, we wanted to incorporate the hash table and hashing functions, and how much it was going to cost us in terms of variable space. Clearly, whilst the hash table speeds up access to data structures in the implementation, its inclusion was likely to slow down an exhaustive exploration of the state space of a prospective model (and involve more memory).

Initially we aimed to include the dcache and hashing functions, but after iterating around the process of representing the variables of interest and estimating state space sizes (see the sequel), we decided to omit it. We decided instead to concentrate on modelling the core functionality of the file system. Where the dcache and hashing functions were used in the implementation (i.e. to find a dentry corresponding to a name), these could be modelled as simple searches across the small pool of dentries available for allocation. Thus it was possible to model the effect of hashing and retrieval and localise the impact of not having a hash table when abstracting the implementation's algorithms.

5.1 Results of Analysis of Data Structures

Having resolved the issues of dynamic memory allocation and the dcache it was possible to stabilise the fields of interest in the structures we needed. Appendix B.1 shows the final tables for superblocks, dentries and inodes that resulted as part of this activity.

For the dentry table, for example:

- *d_flags, d_alias, d_time, d_fsdata, d_cookie, d_mouted* could all be judged low relevance given the methodological scope.
- *d_name* was judged low relevance given that there was another field *d_iname* used to record a shortened version of the name (for hashing purposes). Since we would only need a small number of names, *d_iname* was selected as the field in which to store the name.
- *d_op* and *d_sb* were judged low relevance as the operations on dentries and each dentry's superblock could be "hard wired" in the any model given the scope. I.e. we would only use one set of functions, there would always be only one superblock.
- *d_hash* was considered low relevance in the context of our decision not to model the dcache.
- *d_lock* was judged high relevance. Locking was considered a high priority according to our predetermined scope (important for concurrency).
- *d_inode, d_parent, d_child, d_subdirs* were all judged high relevance, as they capture the structure of the file system (corresponding inode, parent directory, siblings and children respectively).
- *d_lru* and *d_rcu* were problematic. They were eventually assigned relevances of medium and high respectively. Preliminary investigation into their use in the implementation was inconclusive and we were unsure how critical they would turn out to be in the modelling exercise. This was our best assessment at the time[3].

[3] In the end, neither featured significantly in the models produced

- *d_count* was assigned a relevance of high. This was a very important field, recording whether a dentry was assigned (when above 0) and the number of processes accessing a dentry (when above 1). It turned out to be especially important for validating the algorithms abstracted from the implementation.

5.2 Representation and State Space

Once the data structures and fields of interest had been assigned a relevance, the next stage was to select fields, decide on their representation and estimate the potential size of the state space. This was achieved initially by selecting the fields of high relevance; the intention was to review and add lower priority fields if the potential state space was sufficiently low. The first step was to decide a strategy for representing each field, and estimate the number of bits required. Appendix B.2 shows the results of this activity.

For example, considering the dentry table:

- *d_inode* and *d_parent* were assigned 3 bits each, allowing one to reference a maximum of 8 inodes and dentries in the file system.
- *d_child* and *d_subdirs* were allocated 8 bits each, allowing up to 8 siblings and children of a dentry to be marked rather than stored as a linked list.
- *d_iname* was allocated 3 bits allowing for 8 different names (and thus the maximal width of the directory structure).
- *d_count* was allocated 3 bits, allowing up to 2 processes to be accessing a dentry at a time (with space for another 4 processes).
- *d_lock* was initially assigned 3 bits, allowing for 1 bit giving the status of the lock, 1 bit for the process (up to 2) holding the lock, and 1 bit indicating a waiting process[4].
- *d_rcu* was allocated 4 bits[5].

Given the bit allocations it was now possible to calculate the potential information space of a model – that is, the number of states (ignoring control flow) that a model using the data model would exhibit. Knowing that control flow (i.e. abstract program counters) would add a further dimension to this calculation, we aimed to keep this to a minimum; the aim was for a figure of the order $2^{50} – 2^{500}$, based on previous model-checking experience[6].

The number of bits needed for each dentry was, for example, 35. For inodes it was 26. 8 of each were needed, yielding (35+26)*8=488, plus a superblock at 9. That made 2^{497}, which was deemed acceptable – although it meant that including lower relevance data fields was something that would not be possible until after the initial modelling phase of the study.

[4] Retrospectively, the process ID and wait queue (for fairness purposes) were not significant in the models produced.

[5] Retrospectively, d_rcu was not a significant feature of the models produced.

[6] These were the reasonable upper bounds we expected to be able to check. This figure did not include control flow, additional variables etc., but it was a *potential* figure – not every possible data state would be a reachable data state in the model. In addition, the intent was to make the number of nodes in the system a generic parameter to any model, i.e. what we were attempting to estimate was not the actual state space but the upper bound on a spectrum of different potential spaces.

5.3 Identification of Properties of Interest

Once the relevant fields were decided, it was possible to identify specific properties of interest. The aim was that these, in addition to more general properties such as deadlock freedom, would form the basis of the verification exercise. By showing that such properties held at stable points in the model (i.e. between calls to the functions operating on the file system), it would be possible to show that the file system was maintained in a consistent state, and thus infer conclusions about the correctness of the model (and implementation).

The consistency properties identified were of two forms:

- *structural properties*: expressing the static relationships between the information structures of the file system that ought to be maintained by the functions operating on them.
- *reference properties*: expressing constraints on the reference counters in the file system (notably *d_count*). I.e. that the reference counters were maintained correctly by the functions operating on the file system.

Examples of structural properties included the following:

- that the inode referenced by each dentry (in *d_inode*) was assigned[7] and referenced the right dentry (in *i_dentry*)
- that the siblings and children of each dentry (*d_child* and *d_subdirs* respectvely) were assigned and that their parent references (*d_parent*) agreed.

There was one important reference property, which was:

- that the *d_count* was always maintained at a *sensible* value a stable points in the model (i.e. when no process was accessing the file system). The sensible value depended upon whether the model was designed to deallocate unused nodes immediately (then *d_count* then should always be 1 for allocated nodes) or whether it was designed to mark nodes for deferred deletion (*d_count* could also carry a 0 value). Both deallocation strategies were considered. In other words, the property was designed to check that the operations on the file system did not cause the reference count to "drift" over repeated calls.

5.4 Retrospective Comments

At the time of writing, has only been possible to go through the process of abstracting the information structures once. Future work should certainly include a review of this phase of the study. In hindsight, most of the decisions taken at this point worked well but a few did not. For example, the decision was taken not to represent the inode reference count (*i_count*). The rationale behind this was that because of our initial scope there would be a one to one relationship between dentries and inodes in the system. This meant that we ought to be able to use the dentry reference count (*d_count*) as a surrogate value for *i_count*, since they ought always to be the same. However, this made the code abstraction and modelling phases more difficult, as disentangling the inode logic (involving *i_count*) from the dentry logic in

[7] "Assigned" meaning part of the file system, i.e. memory allocated and marked as in use (*d_count*>0).

at least one place became problematic (due to concurrency considerations). In addition, we eventually ended up with several redundant data fields in the models produced.

Another aspect of this phase of the study that was less effective than hoped was the use and relevance of the consistency properties identified. One of the reasons for this was simply that, given constraints in resources, we have not made as much progress as we would have liked adding the consistency properties to our models. However, another reason stems from the way the static relationships are maintained by the implementation, and this aspect was unforeseen. When nodes are added or removed from the file system the new static relationships are computed locally for the parent node (and children) affected by the change. However, this is not achieved by modifying the existing relationships – instead the new relationships are computed from scratch from the parent (*d_parent*) references (c.f. the *update_parent()* function in pseudocode and models below). Our view was that this devalued the use of certain structural properties as a basis for verification. In particular, it would perhaps have made more strategic sense to abstract the algorithm computing the relationships in question, and verify it independently from the main file system model. This would have achieved much the same result as including structural properties in the main model, but would have been a more efficient way of working.

6. Abstracting from the Linux Code

Despite our initial hope of automating the process of abstracting a model from the VFS implementation that correctly reflects the way the VFS handles the creation and removal of files and directories, it turned out to be one of the most difficult parts of this case study.

The main reasons for these difficulties can be found in the size of the VFS implementation, the heavy use of dynamic memory allocation and the utilisation of function pointers. Furthermore, concurrency issues – an important focus for our modelling approach – contributed a great deal to the number of model revisions that were required in order to eliminate errors.

6.1 Modex

Back in 1999, Holzmann and Smith [HS99] developed the tool Modex. It can be used to mechanically extract high-level verification models for SPIN from implementation level C code. In order to do so, Modex requires a user-defined test-harness guiding it through the implementation under consideration. As presented in the previous section, we already knew which operations and which bits of the VFS data structures we are interested in, enabling us to define a test driver and to run Modex on the VFS code.

Unfortunately the tool failed parsing the kernel source. We unsuccessfully experimented with unmodified and preprocessed source code. Our assumption is that Modex cannot deal with those fragments of the Linux source that do not comply with the ANSI C standard or contain compiler-dependent code.

6.2 Pseudocode

After the realisation that automatically abstracting a model from the VFS sources was not possible, we decided to manually inspect the code in order to identify the functions operating on those parts of the VFS data structures we were interested in. The goal of this step was to provide some abstract pseudocode of the VFS, which could then be translated into models for SPIN and SMART. We decided to produce pseudocode in a C-like syntax because this would be fairly close to the original code as well as to a Promela model for SPIN.

Although the authors are familiar with the internals of the Linux kernel and the development of Linux device drivers, none of us had prior experience with the VFS and the file system infrastructure. In order to avoid doing a completely manual analysis of the code, which would require us to manually follow function pointer calls and to perform macro-expansion, we decided to generate call traces into the running kernel in order to get an impression about what is done in which order. Our initial hope was to be able to automatically extract a basic model of the locking operations used in the system calls listed below:

- *mount()*: Mount a file system. Since we are not modelling the physical storage, we were mainly interested in understanding what the in-memory view of an empty, freshly mounted file system is.
- *umount()*: Not used.
- *creat()*: Create a file.

- *open()*: Open an existing file (not used) or create new file. Contains the actual logic for *creat()*.
- *close()*: Close file opened with *open()*. Not used.
- *unlink()*: Remove a file.
- *mkdir()*: Create a directory.
- *rmdir()*: Remove a directory.

Those system calls marked as “not used” were initially considered as important. However, in the end we decided not to include them into the models in order to avoid extending the model’s state space by introducing per-process lists of open files and an abstraction of the physical medium as well as an on-disk representation of the VFS data structures.

To obtain function traces from a running Linux kernel we adopted the *KFT*[8] tool to work with Linux 2.6.18 and implemented a few simple test drivers that initialised *KFT* for a particular system call, did the call in respect of a separate file system used in our experiments, and obtained the trace. *KFT* itself employs the *finstrument-functions*[9] capability of the compiler to add instrumentation callouts 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 could be translated into the respective function names. The following listing represents an excerpt of the call trace for the *creat()* system call. Addresses have been translated into function names and most functions related to permission checking as well as operations on the specific file system layer and the physical device have been removed for the sake of simplicity:

```
sys_creat
| sys_open
| | getname
| | | kmem_cache_alloc
| | | strncpy_from_user
| | get_unused_fd
| | | find_next_zero_bit
| | | expand_files
| | filp_open
| | | open_namei
| | | | path_lookup
| | | | | link_path_walk
| | | | | | __link_path_walk
| | | | | | | permission
| | | | | | | do_lookup
| | | | | | | | __d_lookup
| | | | | | | | __follow_mount
| | | | | | | | | lookup_mnt
| | | | | | | | | dput
| | | | | | | | | | _atomic_dec_and_lock
| | | | | | | dput
| | | | | | | | _atomic_dec_and_lock
| | | | | | | permission
| | | | | | dput
| | | | | | | _atomic_dec_and_lock
| | | | __lookup_hash
| | | | | permission
```

[8] Kernel Function Trace, c.f. http://tree.celinuxforum.org/CelfPubWiki/KernelFunctionTrace
[9] c.f. http://gcc.gnu.org/onlinedocs/gcc-4.2.2/gcc/Code-Gen-Options.htm

```
| | | | | cached_lookup
| | | | | | __d_lookup
| | | | | | d_lookup
| | | | | | | __d_lookup
| | | | | d_alloc
| | | | | ext2_lookup
| | | | | | d_instantiate
| | | | | | | dummy_d_instantiate
| | | | | | d_rehash
| | | | | | | __d_rehash
| | | | vfs_create
| | | | | permission
| | | | | dummy_inode_create
| | | | | ext2_create
| | | | dput
| | | | | _atomic_dec_and_lock
| | | | may_open
| | | | | permission
| | | dentry_open
| | | | get_empty_filp
| | | | get_write_access
| | | | file_move
| | | | generic_file_open
| | | | file_ra_state_init
| | fd_install
| | kmem_cache_free
```

As can be seen, the trace gives an excellent overview of the control flow inside the VFS implementation. It can immediately be noticed that the main logic of *sys_creat()* is in *sys_open()*, which calls *path_lookup()* in order to traverse the file hierarchy up to the point at which the file is supposed to be created, and then invokes *vfs_create()* performing the actual file creation. We can also see some of the locking related functionality, namely the

```
dput
| _atomic_dec_and_lock
```

calls. However, the view of the VFS we obtained from call traces is incomplete and a great deal of effort had to be spent in manual code inspection. The main reasons for this are as follows:

1. The call trace does not reveal what a particular function does on those parts of the VFS data structures we are interested in.
2. Several important function calls are missing in the trace. That is because some functions could not be instrumented because they are supposed to be called from an atomic context in which performing blocking I/O operations (i.e. writing out a log message) is not permitted.
3. Macros were not instrumented.

The final pseudocode was developed in respect of the decisions taken during scoping (see section 4) and data abstraction (see section 5). In particular:

1. The models were supposed to have a fixed number of inodes and dentries. We did not have "real" allocation and deallocation of these structures.

2. Each inode was assigned to at most one dentry. We supported neither symbolic nor hard links.
3. We did not have an underlying specific file system. Hence, “negative” dentries were only allowed in the intermediate steps of the algorithm and non-empty inodes without a respective dentry were ruled out.
4. We did not model the hash bucket used by the dcache.

Most of these decisions contributed to simplifying the pseudocode by allowing us to exclude implementation code handling exceptions and corner cases. However, due to the huge amount of source code to analyse, and the complexity caused by concurrency handling and the use of macros in order to refer to the current process’s context, we sometimes ended up abstracting the intent rather than the implementation itself. While this process allowed us to abstract the core behaviour of the VFS in about 3k lines of pseudo code, instead of the 70k LOC of the implementation, it turned out to be tedious and error prone. Hence, several errors were introduced in the first versions of the pseudo code, which could only be identified later on in the verification process. For each error, the model in which the error was identified was compared with the pseudocode, and then the pseudocode was checked against the implementation, in order to locate the source of the inconsistency or deadlock.

The following code represents our abstract view of the *creat()* function. For the sake of simplicity we decided not to model it as a special case of the *open()* call. All the data types used in the code section are basically equivalent to the type declarations found in the Linux headers. However, those parts of the structs we did not consider important have been removed. Furthermore, we aimed to restrict the usage of pointers and cast operations to an absolute minimum. In some cases they are still required in order to make the pseudo code compile, which is a very helpful feature in order identify type inconsistencies before actually transcribing the pseudo code into a model for SMART or SPIN.

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_creat.c,v $
 * $Revision: 1.11 $, $Date: 2007/11/09 17:51:15 $
 */

/* sys_creat is actually a specific behaviour of sys_open() */
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); }
```

The full pseudocode for the system calls *sys_creat()*, *sys_unlink()*, *sys_mkdir()*, *sys_rmdir()* and *sys_rename()* are given in appendices C.2 to C.6. As shown in Appendix C.1, we also provide pseudo implementations or at least prototypes for various additional VFS functions such as *path_lookup()* or *path_release()*, as well as for functions that belong to other parts of the kernel's infrastructure. Examples are the mutex handlers *up()* and *down()* and the spinlock interface provided by *spin_lock()* and *spin_unlock()*. Another important thing to point out is that our data structures contain additional fields such as the *is_allocated* field in the *dentry* and *inode* structures. These are required in order to designate a particular *dentry* or *inode* as allocated or released in the absence of real allocation, deallocation and pointer references. Respectively, functions such as *allocate_dentry()* or *allocate_inode()* are supposed to find and return data structures in a fixed-size array of dentries or inodes, for which the *is_allocated*-flag is not set. As a result of this, running out of free inodes or dentries have to be valid end states in the resulting SPIN and SMART models.

The pseudo code presented in Appendix C, which was derived from the implementation by manual inspection, has been evaluated by extensive reviewing and cross-checking against the implementation. This is the highest level of confidence that can be gained in the pseudo code, in the absence of tools that can automatically synthesise models from Linux kernel source.

7. The SPIN Model

Constructing the Spin model was essentially a two-stage process. The first step was to produce the core of the model on top of which the pseudocode functionality could be constructed. This involved defining the data structures and equivalent of dynamic memory allocation. Once the core of the model was in place the pseudocode was carefully transcribed into the Promela syntax. This involved taking into account the details of the interface to the core model as well as certain stylistic choices (see below). The full model can be found on-line[10].

7.1 The Core of the Model

The first step in constructing the core of the model was to translate the data structures discussed in section 5 into the Promela syntax. Examples of these (for dentry and inode) are shown in Appendix D.1. The structures derive directly from the information given in appendix B.2 (information modelling). There are two additional type definitions, other than those relating to section B.2 and these concern the model of dynamic memory. The definitions for *dentrypool* and *inodepool* introduce the mechanism by which dentries and inodes are allocated. Each has an array of size *NoofNodes*, the generic parameter used to limit the maximum number of nodes in the file system (8 or less), of structure required (dentry or inode). Each also has a bit array *available* (size 8) to record whether the corresponding structure is allocated or not[11].

Once the data structures were defined it was possible to model the basic mechanisms for allocating and deallocating dentries and inodes. Two examples, the functions for de/allocating dentries are given in Appendix D.2. Note that the variable *dep*, used as a parameter to the inline, is (expected to be) a structure of type dentrypool. Note also that the de/allocation functions are primarily defined in terms of *d_step*s, meaning that they are treated as atomic functions in the statespace construction[12]. Note finally that if an error occurs in the allocation process, meaning that all the nodes are already allocated, the allocation function jumps to *end*. *end* is defined as a valid end state, meaning failure to allocate is not treated as an error in the model.

The final stage in constructing the core of the model was to add the functions required by (but not defined in) the pseudocode, or needed by the test harness. These included, for example, low-level file system operations, such as for allocating and initialising dentries and inodes (*allocate_dentry()*, *allocate_inode()*), initialising the superblock (*init_superblock()*), and finding named dentries (*modelfinddentry()*). They also included low-level functions for manipulating path names (*is_prefix()*, *concat_element()*, *prepend()*, *last_component()*). Two examples: *allocate_dentry()* and *modelfinddentry()* (which supersedes the use of the dcache hash table as discussed in section 5) are included in Appendix D.3. *allocate_dentry()* is straightforward; it

[10] http://www.cs.york.ac.uk/~andyg/filesystem/spinmodel.pml

[11] Note that an alternative model of the data structures was explored in which the information was bit-packed in order to reduce the size of the state vector (and thus memory used in model-checking). However, the bit-packed version did not produce significant savings in state vector size when used in conjunction with the SPIN “compression” option. It also required some processing overheads (model complexity) in the storing and retrieval functions. The approach was abandoned early in the modelling process.

[12] This reduces the size of the state space as intermediate points in the *d_step* are not represented. It also allows hiding of the local variables, which reduces the overall size of the state vector.

allocates a node using *alloc_dentry(),* sets the parent and name (both inline arguments) and sets the remaining fields to their default values (e.g. no children or siblings). *modelfinddentry()* searches the dentry pool for a dentry with a specified parent and name. Note that two assertions are employed to associate constraints with the points in the model they are expected to hold. These are employed throughout the model and their use is described in more detail below. For example, if the function is employed when the parent node is Null valued (equal to the *NoofNodes* parameter) then the dentry name being looked up should be root (which has a unique name 0 corresponding to "/").

7.2 Transcribing the Pseudocode

Transcribing the pseudocode involved translation of the algorithms into the Promela syntax whilst taking into account the interface to core of the model plus a few stylistic considerations (see below). The first stage was to transcribe the supporting functions as given in appendix D. These included: *down(), up()* (for mutexes); *spinlock_lock(), spinlock_unlock()* (for spinlocks); *dget(), dput(), path_release()* (for maintaining *d_count*); *is_directory()* (to establish the status of a dentry); *get_dentry()* (for looking up named dentry); and *update_parent()* (for deriving the new child/sibling relationships). With these functions in place it was then possible to transcribe the *path_lookup()* function, which was integral to every system call we were modelling.

The functions *get_dentry(), update_parent() and path_lookup(),* the most important of the supporting functions, are given in appendix D.4.

get_dentry() is the supporting function which looks up a dentry for a specific parent and name. It interfaces to *modelfinddentry(),* given in D.3, the model-specific version of the look up process described above. Note the use of the assertion to check the value of *d_count* – this is discussed in further detail below.

update_parent() is the function which calculates the new child and sibling relationships for a particular parent node in the file system. It works by scanning the dentry pool for nodes having the correct parent, constructing a child and sibling list as it goes. Then, in a second pass, the parent's children field and its children's sibling fields are updated accordingly.

path_lookup() is the key supporting function. It is used to find a dentry corresponding to a particular path name as well as its parent in the path, and is the basis of all the system calls we modelled. If the function is successful, it returns the node and its parent, and has the side effect of increasing the *d_count* on the nodes returned (reflecting that they are being accessed). The reason why the logic of *path_lookup()* is so elaborate is the number of cases that need to be treated as distinct. – for example the treatment of root, which returns a null parent. There are also additional branches dealing with, for example, next item in path found/not found, end of path reached etc. Note also that one of the parameters to *path_lookup()* is a *cwd*[13] that is prepended to the supplied path in the case where the first item in the path is not root. Although this facility was used at various points in the evolution of the model, it was not required by the final test harness.

The second and final stage of the transcription process was to incorporate the system call pesudocode into the model. For example, the Promela function for *sys_creat()* is given in appendix D.5. *sys_creat()* is probably the least complex of the

[13] I.e. current working directory.

system calls. However, like *path_lookup()*, it is complicated by the number of distinct cases that need to be considered. It begins by using *path_lookup()* to retrieve the dentry of the file to be created (if it exists) as well as its parent (if it exists). If the parent exists it obtains its associated mutex. Otherwise it returns an error (with some special handling of the root case to redecrement its *d_count* i.e. *dput()* it). There are several cases to consider if the parent exists:

- its child exists and is not a directory: in this case the *d_count* of parent and child are redecremented using *path_release()* and the mutex is released. Note that this is not treated as an error case because *sys_creat()* has been derived as a special case of *sys_open()*.
- its child exists and is a directory: in this case the *d_count* of parent and child are redecremented using *path_release()* and the mutex is released. The error flag is set to indicate an error.
- the child does not exist: in this case the file is created by spin locking the dcache, extracting the last element of the path (the file name), allocating a dentry with the appropriate name and parent, incrementing the dentry's *d_count*, locking all the inodes, allocating a new inode and associating it (in both directions) with the dentry, releasing the inode lock, updating the child and sibling links for the parent and its children, decrementing the parent and child's *d_count* (using *path_release()*), unlocking the dcache and releasing the parent's mutex (obtained earlier if the parent exists).

Note that *sys_creat()* also contains a couple of assertions. These are discussed in the next section.

7.3 Stylistic Considerations

There were several stylistic considerations (or design decisions) taken into account whilst constructing the SPIN model. The most important of these were the use of assertions and the design of the variable space.

The implementation (and therefore pseudocode) does not contain assertions – it would be unwise to include code that might halt the operating system. Instead a more defensive style of programming is employed. Properties tend to be checked and appropriate action taken in either case, where they hold and where they do not. This has two effects: firstly it makes the implementation robust against errors, and secondly, the generic nature of functions designed this way can simplify the algorithms.

Good example of this style can be seen in the pseudocode for *dget()* and *dput()* (see appendix C.1), where the value of *d_count* is checked before increment or decrement. This is to guard against a dentry becoming accidentally reassigned (respectively deassigned). However, the logic was also exploited at least once in the pseudocode of one system call, which called *dput()* in a situation where it would possibly have no effect. This simplified the algorithm slightly.

Conversely, when model-checking we are happy for the system to "halt" – it draws our attention to potential problems in the system being modelled (if any exist) and, more importantly, helps establish the validity of the model. For this reason we embraced the use of assertions in the SPIN model.

Assertions were added to the SPIN model during the transcription process from the pseudocode. This meant that the transcription was not merely a syntactic translation of the pseudocode in a number of ways:

- Assertions were added in situations where certain conditions were expected to hold. These included situations where the logic of the pseudocode indicated an assertion ought to hold, as well as some model-specific situations such as correct use of the interface to the core of the model
- Assertions were added to replace some "defensive style" *if* statements such as in *dget()* and *dput()*.
- *if* statements were added in places to guard the use of functions where "defensive style" *if*s had been replaced by assertions. E.g. where *dput()* was used.

Examples of first kind of assertion can be seen in the functions *modelfinddentry()* (appendix D.3), *update_parent()* (appendix D.4) and *path_lookup()* (appendix D.4). *modelfinddentry()* contains core model interface assertions – that the dcache is locked and that if the parent is null the name is root (0). *update_parent()* contains a logical assertion that *update_parent()* is only called with a directory as an argument. *path_lookup()* contains the logical assertion that after the *cwd* has been prepended to the path the path must start with root (0). Examples of the second kind of assertion are in *get_dentry()* (appendix D.4) and *sys_creat()* (appendix D.5). *get_dentry()* contains an assertion which replaces the check in the pseudocode (see appendix C) that *dget()* succeeded. *sys_creat()* contains two assertions checking that the allocation of a dentry and inode succeeded, replacing checks in the pseudocode – this is possible because of the way the model treats failure to allocate as successful termination.

The model currently contains 50 or so such assertions amounting to approximately 3% of the model.

The other important stylistic consideration was the design of the variable space of the model. SPIN has two kinds of variable, global and local to a process. Additionally, global variables may be hidden, when they only appear in *d_step*s so that they are elided in the state vector (reducing the memory requirements for model-checking). There are no local variables at the "function call" level. This is because function calls are modelled by "inline" constructs, which essentially bind interface variables to their point of reference and replace that point of reference with a suitably modified version of the inline text. Because of SPIN's restrictions, and the importance of keeping the size of the state-vector to a minimum, the design of the variables space (what they are, how they're used) is very important. The following describes our approach. Note that it should be possible to optimise the use of variables further.

There are four kinds of variable in the model:

- Global: these are global SPIN variables and are in scope everywhere in the model – for example *dep* (*inp*), the dentry pool (respectively inode pool) from which nodes are allocated.
- Scratch variables: these are hidden global SPIN variables only used within *d_step* code and therefore not appearing as part of the state-vector. These are used in various low-level functions (such as in path manipulation) and functions supporting the test harness.

- Local inline parameters: these are SPIN local-to-process variables used locally within inline functions. They are not declared locally (as this is not permitted), but supplied as parameters to each inline function.
- Test harness variables: these are local-to-process variables used in the body of the test harness.

The local inline parameters required the most thought. A pool of local variables was declared as part of the process body (the test harness). Interface variables to the inline functions were partitioned into levels according to the usage hierarchy e.g. *lvplus1_1* corresponded to a local variable of size one for use in the next level, and "passed" from function to subsidiary function. Variables that had clear use e.g. *plulv_4_1*, for use in the *path_lookup()* function were named mnemonically. Designing the variable space in this way allowed a weak form of encapsulation of local variables, centralised control of which variables were used where to the process body, and meant that variables could be reused from function to function minimising the number of variables required. It also easily allows inlines to be used in more than one concurrent process over distinct local variables.

7.4 Consistency Properties and Concurrency

So far, due mainly to time constraints, only one consistency property is checked by the model. This concerns the value of the *d_count* reference count, which ought to have a value of 1 for allocated dentries when the system is in a stable state (between system calls). The property is stipulated as an assertion in a monitor function that checks the state of the dentry pool (and prints the information during simulation) in between system calls: *printdentries().*

The fact that more consistency properties have not been included is unfortunate, but is tempered by the inclusion of a great many assertions, which was not anticipated at the outset. Also, as mentioned above (section 5), the implementation details brought into question the use of certain consistency properties as a mechanism for checking correctness. Thus, a review of this aspect is needed before more consistency properties are included.

The ability to perform checks on *d_count* is in part due to the absence of concurrency in the model, which simplified the identification of stable points in the system's evolution (when no file system operations were in progress). Although concurrency was a key aim, and the model was designed with concurrency in mind, a concurrent test harness has not yet been produced for the SPIN model. This was due to setbacks in the modelling phase and resource constraints.

7.5 The Test Harness

The test harness is the part of the model that "drives" the system calls. Its purpose is to initialise the file system and then run the system calls in a way that explores all its possible states. The test-harness has two roles:

- *Simulation*, which produces textual output and allows the user to interact with the model. User interactions usually take the form of resolving (apparent) non-determinism in the model, letting the user *guide* the model into interesting states and *inspect* the model's behaviour in those states. It is used to validate the model.

- *Verification,* which produces no textual output (other than the results of the verification) and does not allow user interaction. It explores all reachable states and checks that nothing undesirable happens along the way (such as an assertion being violated).

The file system test harness was designed with the two roles in mind. However, simulation was hampered by the large number of interactions needed to guide the model into some particular state – navigation became extremely difficult[14]. The solution was to implement user interaction directly in the model using the *stdin* channel (keyboard read). A #define[15] was used to switch-out the Promela code relating to *stdin* (the keyboard reads and use of the results in branch conditions) for verification purposes. This can be seen in the test harness body shown in appendix D.6)

The test harness works by implementing a "mock" *cd* function. This is not a *cd* function relating to the kernel (altering *cwd* and reference counts such as *d_count*), but instead a simple way of manipulating the path argument(s) supplied to the system calls. It allows the user to change the source path (supplied to every system call) and destination path (supplied to the rename call) each time, by returning to root, moving down a specified directory, moving up a directory, or staying in the same place (skip). Two functions, *cd()* and *choose_id()*, supporting the manipulation of paths are shown in Appendix D.7, along with one of the monitoring functions *printdentries()*. The latter contains the *d_count* assertion (consistency property) mentioned above.

Once the source (and destination) path(s) have been set, the user may choose which system call to invoke and the model reports success or failure. Finally, they can choose whether to inspect the current state of the system (using the monitoring functions such as *printdentries()*). When the –D myverif option is set for verification purposes all these choices are made on a purely non-deterministic basis.

7.6 Preliminary Results

The simulation phase consisted of a sequence of random tests (to increase confidence in the validity of the each system call as it was added), followed by a sequence of structured tests (once all calls had been integrated into the model). Approximately 100 tests were performed (and reperformed) during the structured testing phase. They attempted to cover all key scenarios (both successful and unsuccessful) for a maximum width of 2 nodes, down to depth of 3 nodes. Both types of testing produced deviations from the expected behaviour. Occasionally these deviations were due to errors in the model (e.g. caused by inadequate protection of assertions) and had no bearing on the validity of the pseudocode. However, in several cases the errors were directly traceable into the pseudocode and implied that the pseudocode or implementation itself were problematic. In each case, where the validity of the pseudocode was called into question, the abstraction process was double checked and found to be in error. The pseudocode was revised accordingly.

Once all of the structured tests were found to succeed as expected, the verification phase was performed, which involved running the verifier on models with various maximum file system settings (*NoofNodes*). Each verification was run a 1.9 GHz machine, on top of cygwin, running on windows XP. The maximum memory

[14] The main reason for this was SPIN's insistence on prompting the user to resolve non-determinism even when no actual non-determinism existed, i.e. in most situations where the Promela code branched.
[15] Normally supplied as a –D option to the verifier.

available was set to 950Mb (in the context of 1Gb RAM), other SPIN options included: exhaustive search (rather than bit hash), state vector compression, partial order reduction[16]. Preliminary results are as follows: The model-checker succeeded checking at a level of 3 and 4 nodes – no further errors were found. The model-checker quickly ran out of memory for 5 nodes. Specific results for 4 nodes are as follows:

Approx Time:	2 minutes
Approx Memory Used:	700Mb
State Vector:	356 byte
Compressed State Vector:	39 byte (+12 overhead)
Compression Ratio:	13%

In addition, several parts of the model were reported as unreachable. On inspection, this appeared to be due to the branches associated with concurrent behaviour, which were never taken due to the sequential nature of the test harness. Attempts were made to analyse 5 and 6 node systems using the bit hashing (non-exhaustive) option. Results were obtained for 5 nodes, with a state size estimate of 7500, but these were of little significance (hash factor 8-9). At 6 nodes, we were unable to achieve any results even for bit-hashing – verification runs did not succeed due to memory problems.

7.7 Retrospective Comments

One of the most important contributions of the SPIN modelling was the model testing activities. The provided important support for the code abstraction process, trapping many errors prior to the verification runs. Due to the scheduling of the work, this turned out to be to the benefit of the SMART modelling phase, which was based on later versions of the pseudocode. This in turn meant that the SMART model was able to concentrate more on validating the behaviours particular to concurrency (see next section).

The preliminary verification results were not so useful. The verifier only ran successfully up to 4 nodes, and it was unsurprising that no errors were found given that the testing phase has already explored most (if not all) distinct scenarios relating to a 4 node file system. The verification problems were caused principally by memory shortage, which highlights the significance of minimising the complexity of the model. To this end, future work will involve optimising the variable space, test harness and SPIN options, as well as running the verifier on a machine with more available memory.

Other important items of future work include provision for multiple processes, and a review – and incorporation – of consistency properties. Adding processes and consistency properties will place more of a burden on the memory requirements for verification. However, it is hoped that the use of *partial order reduction* techniques, in the presence of concurrency, will alleviate some of the overheads.

[16] However, the model contained no concurrency so this probably had no effect.

8. The SMART Model

The translation of the pseudocode into SMART had to comply not only with the specifics of the SMART modelling language (Petri nets) but also with other restrictions imposed by the need to apply the most advanced symbolic model checking techniques, which are only available under additional constraints. One such constraint is the Kronecker consistency requirement which, informally, demands that Petri net constructs that are functionally dependent on each other (Petri net places) be grouped in the same partitioning subnet.

SMART is designed as a tool for logical and stochastic analysis of concurrent systems. Modelling software is not the main target of this language. For this reason, for the translation of the file system pseudocode, we had to manually introduce program counters into the Petri net. Other limitations of the SMART language (no data structures, no recursion, no dynamic memory allocation) made the task more challenging, but did not ultimately hamper the model development.

8.1 Related Work Specific to SMART

From our experience, the VFS model ranks with the most complex systems ever modelled in SMART. This perspective is not reflected solely by the shear size of the model (over 2600 lines of SMART code), but also by the inherent complexity of the system itself. For comparison, two other similar industrial-size applications modelled in SMART are:

- NASA's Runway Safety Monitor [SC07]: a protocol for detecting incidents on airport runways. The model is parameterised by the number of aircraft (called targets) that are monitored, each aircraft being represented by its 3-D (discretised) coordinates and flight status. The SMART file is 1850 lines long. The state-space exploration takes under 5 minutes for the smallest relevant set of parameters (1 target, 3x3x3 position grid).

- NASA's clock synchronization [Min93] and self-stabilization protocols [Mal06] for the SPIDER fault-tolerant architecture. The protocol is parameterised by the number of nodes, clock wrap-around period, and other protocol related data. The SMART file size is 1190 lines. The protocol can be instantiated under various fault assumptions, ranging from benign to symmetric and Byzantine. The smallest relevant setting is for 4 nodes (3 good nodes and 1 faulty). The tool is not able to build the state-space for the most complex setting (Byzantine fault, fully randomised initial state) before running out of memory (on a 16GB machine), but is still able to analyse partial configurations.

8.2 Model Components

The process of extracting the model variables was similar to that for SPIN. The abstract model has is parameterised by:

- The (maximum) number of dentries (ND)
- The (maximum) number of inodes (NI)

- The number of concurrent processes (NP) making calls to the file-system (this can also be viewed a parameter not of the file system itself, but of the test harness for the SMART model);

In principle, the model variables are represented as Petri net places, and instructions are represented as Petri net transitions. Since the model is parameterised, the fields of the dentry and inode data structures are represented as arrays.

```
for (int i in {1..nd}) {
  place
    d_allocated[i],   /* is allocated? flag */
    d_parent[i],      /* id of parent: 0=n/a, or 1..ND */
    d_count[i],       /* reference count */
    d_lock[i],        /* not used */
    d_inode[i],       /* id of corresponding inode: 0=n/a, or 1..NI */
    d_subdirs[i];     /* number of subdirectories */
...
}
```

This is because the SMART modelling language does not support records. The SMART code is nevertheless quite readable, as the i-th dentry is simply represented as the collection of all the i-th elements in the above arrays.

The convention adopted for locks and mutexes is to have a positive value representing “available” and zero for “not-available”. Hence, acquiring a lock/mutex removes a token form the Petri net place storing the lock value, while releasing a lock adds a token back to the place.

Additionally, the model has to instrument a program counter (instruction pointer) for executing the four file operations (create and delete file, make and remove directory). There is choice of modelling the program counter as an integer variable (hence a Petri net place in our model) or, equivalently, as an array of Boolean variables. We opted for the latter approach.

8.3 Modelling Restrictions in SMART

Path Names

We had to circumvent other limitations imposed by the simplicity of the SMART modelling language. This lack of sophistication is a benefit in many circumstances, but in the case of software, the modeller is forced to get creative. One such situation is posed by the need to represent the tree structure of the file system. Lists are not supported in SMART, therefore we had to adopt an abstraction mechanism that does not impair the ability to perform a logical analysis of the original (not abstracted) system.

From the logical point of view, operations with fully qualified filenames (path+filename) only test for the path name being identical or not with an existing one. As long as each type of operation is represented in our model, the abstraction is valid and offers full coverage all possible relevant situations.

For example, in the abstract model, we represent each fully qualified pathname with an integer. Initially, the value 1 is reserved for the root (‘/’) and value 2 for the folder ‘/lost+found’ (created at mount time). Any (distinct) file/folder that is created subsequently is assigned an abstract index. The call to create a dentry is of the form:

create(Dentry_idx file_id, Dentry_idx int parent_id), where Dentry_Idx is the type [1..ND] (ND = size of dentry array)

In the initial state illustrated above (‘/’ and ‘/lost+found’), the next legal create calls can be only *create(i, j),* with i in [3..ND] and j in [1..2]

- *create(i,1),* with i>2, corresponds to adding a child to '/' with any other name except 'lost+found'
- *create(2,1)* represents the attempt to create '/lost+found' again; which should behave accordingly, i.e. does not create a new dentry
- *create(1,1)* represents the attempt to create the root again; should have similar outcome: denied

Similarly for the calls create(i,2) (create children of '/lost+found'):

- *create(1,2)* is illegal
- *create(2,2)* is illegal
- *create(i,2)* with i>2 is valid

Besides,

- *create(i,j)* with j>2 represents an attempt to create a file in a non-existent path

Following a successful *create(i,j)* (say we requested *create(4,1),* which corresponds to *sys_creat(string s),* s different than '/' or '/lost+found'), we have a new abstract string present in the system. For clarity, let's assume that '/a' was created. Therefore, the index 4 represents the new abstraction, given by the equivalence relation: idx==4 iff string=='/a'. The new system state is: 1='/' 2='/lost+found' 4='/a'.

The next valid create request is of the type *create(i,j)* with i not in {1,2,4} and j in {1,2,4}. For example, to create file 'b' as a child of root, we might call:

create(7,1) // – 7 may be replaced by any integer other than 1,2,4 but to create 'b' as child of '/a' we would call *create(7,4)*

On the other hand, trying to *sys_creat('/a')* again is still represented by *create(4,1),* while trying to sys_creat 'a' anywhere but in the root is a valid call, which could be

- *create(7,4)* // if trying to *sys_creat('/a/a')*

or

- *create(7,2)* // for *sys_create('/lost+found/a')*

If '/a' is removed, then index 4 becomes available for any other distinct string not already in the system. Creating a file to take its place, means we have introduced a new abstraction function: idx==4 iff filename==<the_new_string>

8.4.Illustrative Code Snippets

Below is a code excerpt, illustrating a particular instruction (step number 8) in the pseudo-code of the *create()* routine, which tests for the file argument passed to the routine to be allocated and not be a directory. There are two transitions, one for the “then” and one for the “else” branch, respectively. Depending on whether the guard is satisfied or not, the instruction pointer is moved either to line 9 or line 12 of the code. No other state changes are performed by this step.

```
/* ---------- Create step 8 ---------- */
// --- if (file.is_allocated && is_directory(file))
for (int i in {1..nd}) {
    trans
      t_create_step8_then[p][i],
      t_create_step8_else[p][i];
    arcs(
      p_create_line8[p]:t_create_step8_then[p][i],
        t_create_step8_then[p][i]:p_create_line9[p],
      p_create_line8[p]:t_create_step8_else[p][i],
        t_create_step8_else[p][i]:p_create_line12[p]
    );
    guard(
      t_create_step8_then[p][i]:
       tk(p_file[p])==i &   tk(d_allocated[i])>0 & tk(d_subdirs[i])>0,
      t_create_step8_else[p][i]:
       tk(p_file[p])==i & !(tk(d_allocated[i])>0 & tk(d_subdirs[i])>0)
    );
}
```

Another example, that does transform the state of the system, is illustrated below. Step 13 in the *create()* routine allocates dentry #i as a child of d_entry #j. To that extent, it sets the d_allocated[i], d_count[i] and d_lock[i] to 1, and the value of d_parent[i] to j. The transition guard enforces that d_entry #i should not be already allocated. After setting these values, the instruction pointer is moved to line 14 in the *create()* procedure.

```
/* ---------- Create step 13 ---------- */
// --- allocate_dentry
for (int i in {1..nd}) {
  for (int j in {1..nd}) {
    trans
      t_create_step13[p][i][j];
    arcs(
      p_create_line13[p]:t_create_step13[p][i][j],
        t_create_step13[p][i][j]:p_create_line14[p],
        d_allocated[i]:t_create_step13[p][i][j]:tk(d_allocated[i]),
        t_create_step13[p][i][j]:d_allocated[i],
        d_count[i]:t_create_step13[p][i][j]:tk(d_count[i]),
        t_create_step13[p][i][j]:d_count[i],
        d_lock[i]:t_create_step13[p][i][j]:tk(d_lock[i]),
        t_create_step13[p][i][j]:d_lock[i],
        d_parent[i]:t_create_step13[p][i][j]:tk(d_parent[i]),
      t_create_step13[p][i][j]:d_parent[i]:j
    );
    guard(
      t_create_step13[p][i][j]:
       tk(p_file[p])==i & tk(p_parent[p])==j & tk(d_allocated[i])==0
    );
  }
}
```

The entire SMART file is too large to be listed here. For reference, we have included the portion of the file that encodes the *create()* operation in Appendix E. The full model is available on-line[17].

8.5. Results for SMART

At this stage, the main purpose of the analysis is to determine the tool's ability to handle the large state-space size of the parameterised model. SMART was able to complete the reachable space for reasonable values of the parameters, not only for the trivial cases.

Below is a summary of the state-space generation runs for 1 (table 1) and 2 (table 2) processes.

8.6 Preliminary Results for *Parallelised* SMART

In addition to analysing the file system model using SMART, verification was also performed using a prototype [ELS07] which parallelises the saturation algorithms used in SMART. The prototype, written in C using the POSIX *Pthreads* library [LB98], executes in parallel on a multi-core PC (shared-memory architecture), and can be directed to use 1 (sequential), 2, 3 or 4 cores.

Preliminary verification results are given in table 3. Note that no attempts have yet been made to optimise the file system model for efficient parallel analysis.

The results for the file system case study fit with those of a stereotypical “low parallelism” model. For the parallel FIFO algorithm, only a small (approximately 20%) run-time overhead is introduced on the first core. However, only a small (approximately 5%) improvement is achieved on the second core with diminishing improvements on the third and fourth core. This means that the parallel algorithm is unable to improve over the sequential algorithm on four cores, demonstrating an approximate 10% slowdown. The memory overhead for the parallel algorithm is slightly less than 2x, which is a typical memory increase due to the introduction of upward arcs into the MDD structure, and Pthread mutex locks.

[17] http://www.cs.york.ac.uk/~andyg/filesystem/smartmodel.sm

# params ##D #I	# states	time(s)	mem(KB)
2 2	283	0.54	291
2 3	283	0.59	302
2 4	283	0.65	314
2 5	283	0.70	325
2 6	283	0.77	337
2 7	283	0.82	348
2 8	283	0.87	360
3 2	1086	1.11	802
3 3	2660	2.03	1948
3 4	2660	2.28	2035
3 5	2660	2.54	2121
3 6	2660	2.84	2208
3 7	2660	3.07	2294
3 8	2660	3.34	2381
4 2	3339	2.05	1547
4 3	20395	8.13	8705
4 4	80461	13.42	16776
4 5	80461	15.20	17501
4 6	80461	17.08	18227
4 7	80461	18.94	18952
4 8	80461	20.92	19675
5 2	9406	3.75	2786
5 3	110359	20.75	20532
5 4	951538	52.74	75189
5 5	5604562	76.92	184571
5 6	5604562	85.09	187811
5 7	5604562	92.80	190853
5 8	5604562	101.29	193977
6 2	25163	6.56	4537
6 3	480011	40.97	39674
6 4	6827399	126.21	212000
6 5	87900191	382.34	1430972
6 6	0	0.00	0
6 7	0	0.00	0
6 8	0	0.00	0
7 2	68874	11.24	7074
7 3	1814603	79.24	71333
7 4	37223248	255.54	502398
7 5	0	0.00	0
7 6	0	0.00	0
7 7	0	0.00	0
7 8	0	0.00	0
8 2	162523	19.37	10446
8 3	6228787	145.54	116699
8 4	170672245	507.87	1059037
8 5	0	0.00	0
8 6	0	0.00	0
8 7	0	0.00	0
8 8	0	0.00	0

Table 1. State-space generation results for one process

# params ##D #I	# states	time(s)	mem(KB)
2 2	18934	3.43	4331
2 3	18934	4.02	4488
3 2	485587	93.28	78968
3 3	2992118	1139.61	784659

Table 2. State-space generation results for two processes

(NB: The zeroes mean SMART ran out of 4GB of memory).

Type	Run-time (s) (Cores)				Relative Memory (Cores)			
	1	2	3	4	1	2	3	4
	D=2 I=2 P=1 Sequential: 0.42(s) 65542(b)							
fifo	0.50	0.48	0.47	0.47	1.74	1.75	1.78	1.78
chain	0.52	0.51	0.51	0.51	1.77	1.75	1.75	1.75
	D=3 I=2 P=1 Sequential: 1.69(s) 195300(b)							
fifo	1.88	1.82	1.79	1.78	1.76	1.77	1.78	1.79
chain	1.93	1.92	1.91	1.91	1.78	1.74	1.73	1.73
	D=4 I=2 P=1 Sequential: 3.32(s) 295112(b)							
fifo	3.80	3.67	3.58	3.53	1.77	1.80	1.82	1.82
chain	3.94	3.90	3.90	3.88	1.78	1.76	1.76	1.75
	D=3 I=3 P=1 Sequential: 11.03(s) 887170(b)							
fifo	13.34	13.02	12.88	12.64	1.79	1.84	1.85	1.85
chain	13.57	13.54	13.51	13.50	1.80	1.80	1.80	1.79
	D=4 I=3 P=1 Sequential: 151.22(s) 4957529(b)							
fifo	179.69	168.21	162.10	160.49	1.83	1.88	1.88	1.88
chain	191.20	189.20	188.79	188.74	1.82	1.83	1.82	1.82

Table 3. Run-time and memory results for the file system model.

9. Related Work

While techniques for verifying the correct use of file system interfaces represented as finite state machines are presented in [DLS02] and [DF01], work on verifying properties for file system implementations as shown in this report is quite rare.

In [AZKR04] a correctness proof for a formalised basic file system implementation that uses standard file system data structures such as inodes and fixed-sized disk blocks is presented. It considers data structures which are also covered in our work. By having a notion of disk blocks, it also deals with their respective on-disk representation. In the paper, the implementation is proved correct by establishing a simulation relation between the specification of the file system, formalised as a map from file names to sequences of bytes and its implementation using the Athena proof system. The work done by Arkoudas et al. differs from ours as it does not deal with the verification of a "real" file system implementation and concurrency related issues. Hence, it does not involve the process of abstracting a model from a given implementation and its environment.

Two publications dealing with the verification of actual file system implementations are [YTME04] and [YST+06].

In [YTME04], model checking is used in systematic testing to find errors in the specific file system implementations EXT3, JFS and ReiserFS. Their 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 recover no matter in what order buffer cache contents are written to disk, and a recovery checker using the *fsck* file system recovery tool. The system starts with an initial, empty file system and recursively generates possible successive states by executing system calls affecting the file system. After each step the system is interrupted and *fsck* is used in order to check whether the file system under test can recover to a valid state.

[YST+06] uses a similar approach combined with symbolic execution in order to generate test cases that can be used to crash or exploit a file system implementation.
Both approaches are similar to our work in the sense that they are striving to expose problems in actual file system implementations that can lead to serious inconsistencies or security exploits. However, [YTME04] and [YST+06] are based on runtime verification techniques that cannot exhaustively explore the state space of the implementation. A big advantage over our work is that these techniques do not require the tedious and error prone manual abstraction of a model from the implementation, which was required in our case.

Verification approaches that mechanically analyse the source code of operating system components and that can be used in order to automatically and exhaustively identify property violations are presented in [Hen02], [CC+04a] and [BR01]. In theory these tools are able to prove a file system implementation to be free of deadlock situations due to improper use of locking mechanisms. However, as shown in [ML06], the tools also require tedious manual preprocessing of the original source in order to be able to parse and model check it. According to the experience of the authors, the amount of man-hours required to prepare the VFS implementation for being verified with BLAST up to the same extent as shown in this report, would be equivalent to our approach since similar manual abstractions would be required.

10. Conclusions

This report has outlined our experiences model checking part of a Linux file system. In particular we have described the aims of the project, the domain, the scope and methodology followed, the abstraction of the data structures used by the file system, the abstraction of the Linux code and the construction of two models in SPIN and SMART. These experiences constitute intermediate results in the sense that additional work is now ongoing, with the aim of maximizing the benefits of the study (and return on investment).

Successes

Several aspects of the study can be considered as successes. The abstraction of the data structures, on the whole, worked well – although there were minor difficulties associated with the decision to use *d_count* as a surrogate for *i_count*. Both modelling phases were also largely straightforward, and each can be considered a success in its own way. The SPIN modelling phase was particularly useful, through the use of assertions and structured testing, for validating the sequential aspects of the pseudocode – although the results for verification were hampered by problems with memory requirements. On the other hand the SMART model, which was able to capitalise on SPIN's analysis of the pseudocode, was able to concentrate on validating the concurrent behaviour of the pseudocode. The verification results for SMART were impressive, but limited to features such as deadlock freedom.

Limitations and difficulties

Despite the independent success of the modelling phases there was a key drawback. During the design and construction of the models there was some *drift* in the focus of each, and this meant that comparisons between models were difficult to infer. In particular:

- restrictions in the SMART modelling language constrained the way *path_lookup()* could be modelled, resulting in an abstracted algorithm and less faithful (cf the pseudocode) interface to the system calls. Conversely, the SPIN model adhered closely to the pseudocode.
- the SPIN model was sequential and deallocated nodes as soon as they were unassigned, which was unfaithful with respect to the pseudocode. On the other hand, the SMART model allowed multi-process concurrency, including concurrent a *clean_up()* operation to deallocate unassigned nodes. The difference was attributable in essence to time lost in testing (for SPIN), when pseudocode problems were found and resolved.
- the SPIN model contained many assertions, including one consistency property, whilst the SMART model focussed on concurrency issues, such as deadlock.
- the SPIN model incorporated the *sys_rename()* system call, whereas the SMART model elected to prioritise progress on the other four system calls.

Another significant drawback concerned the SPIN verification results, which were not very useful. Realistically, a 4-node verification in a sequential model does not tell us very much about the correctness of the pseudocode, let alone the implementation.

By far the most severe difficulties concerned the abstraction of the LNUX code, which was the hardest part of the entire exercise. The manual nature of the abstraction and the problems in validating the pseudocode make any scientific conclusions about the correctness of the implementation difficult to infer. Having a faithful abstraction was key to the aim of adding to the existing confidence in the Linux implementation. We believe further research is needed in this area.

Better Tools for Abstraction.

Abstracting a faithful model from the VFS implementation turned out to be one of the most difficult parts of this research project. However, it can also be considered as one of the most common tasks to be done in post-hoc software verification using symbolic model checking or theorem proving. Hence we suggest, that future research should explore this area, enabling the development of automated tools mechanising this process. An ideal tool for the purpose of model abstraction would require two inputs. Firstly the program under consideration, secondly a specification stating precisely which parts of the implementation's data structures and functions, or which verification properties a user is interested in. The tool would than be able to automatically abstract a minimal model of the system in respect of the specification, dynamic memory allocation and concurrency issues. Actually, tools such as BLAST [Hen02] are coming close to this goal by using the CounterExample Guided Abstraction Refinement (CEGAR, cf. [CC+04b]) paradigm. However, to the authors' knowledge all of them have restrictions regarding the input language, memory allocation and concurrency. Especially in the context of the verification of operating system components, restrictions to the programming language accepted by the tool have the highest impact. That is because these software components are usually not written in plain ANSI-C but contain architecture and compiler specific code sections as well as inline assembly. Hence, research on abstracting models from lower-level intermediate code or even object code might be worthwhile.

Future work

The most immediate source of future work concerns obtaining comparisons between the SPIN and SMART models. Presently, the two models are so dissimilar that comparisons are meaningless. In order to bring them into a comparable state the following work will be undertaken, primarily on the SPIN model:

- the SPIN model's system call interface will be brought into line with that of the SMART model. The *path_lookup()* function will be abstracted accordingly.
- Multi-process concurrency will be added to the SPIN model, including a concurrent *clean_up()* operation.
- The assertions will be removed (once shown to hold). Alternatively, assertions might be added to the SMART model as "error" transitions.
- Verification of the two models will be executed in comparable settings (i.e. the same machine, with as much memory as possible).

Other areas of future work may include:

- optimisation of the SPIN data space, test harness etc. for increased verification coverage
- a review of the data structures included in the abstraction
- a review and introduction of additional consistency properties, in line with our original expectations

Potentially, there is still much to do to add to this study. For one, we have not yet iterated as per Figure 3, with the aim of adding functionality (e.g. mounting, hard links) and moving the models closer to the media representation (e.g. modelling the behaviour of EXT2). The preliminary results presented herein represent a first step. We hope to continue the work for some time to come.

References

[Abr96] J-R Abrial, “The B Book – Assigning Programs to Meanings”, Cambridge University Press, 1996.

[AZKR04] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, Martin Rinard, “Verifying a File System Implementation”, LNCS 3308, p. 373-390, Springer Berlin, Germany.

[Bov02] Daniel P. Bovet, Marco Cesati. "Understanding the Linux Kernel", 2nd edition. O'Reilly Media Inc, Sebastopol, USA. 2002.

[BR01] T. Ball and S. K. Rajamani, “Automatically validating temporal safety properties of interfaces”. In SPIN 2001, vol. 2057 of LNCS, pp. 103-122.

[CC+04a] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav, “Efficient verification of sequential and concurrent C programs”. *Formal Methods in System Design*, 25(23):129-166, 2004.

[CC+04b] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, Helmut Veith, “Modular Verification of Software Components in C”, *Transactions on Software Engineering* (TSE), Volume 30, Number 6, pages 388-402, June 2004.

[CJMS06] G. Ciardo, R.L. Jones, A.S. Miner, and R. Siminiceanu, “Logical and stochastic modeling with SMART”, *Performance Evaluation*, 63:578–608, 2006.

[CLM07] G. Ciardo, G. Lüttgen, and A.S. Miner, “Exploiting interleaving semantics in symbolic state-space generation”, *Formal Methods in System Design*, 31(1):63–100, 2007.

[CLS01] G. Ciardo, G. Lüttgen, and R. Siminiceanu, “Saturation: An efficient iteration strategy for symbolic state-space generation”, in TACAS, vol. 2031 of LNCS, pp. 328–342. Springer, 2001.

[CRK05] Jonathan Corbet, Alessandro Rubin and Greg Kroah-Hartman, "Linux Device Drivers". 3rd edition. O'Reilly Media Inc, Sebastopol, USA. 2005.

[DF01] R. DeLine and M. Fähndrich, “Enforcing high-level protocols in low-level software”. In Proc. ACM PLDI, 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle, “ESP: Path-sensitive program verification in polynomial time. In Proc. ACM PLDI, 2002.

[ELC07] J. Ezekiel, G. Lüttgen, and G. Ciardo, “Parallelising symbolic state-space generators”, in CAV, vol. 4590 of LNCS, pp. 268–280. Springer, 2007.

[ELS07] J. Ezekiel, G. Lüttgen, and R. Siminiceanu, “Can Saturation be parallelised? On the parallelisation of a symbolic state-space generator”, in PDMC, vol. 4346 of LNCS, pp. 331–346, Springer, 2007.

[Eth05] http://vstte.ethz.ch/

[Hen02] T. A. Henzinger et al, “Temporal-safety proofs for systems code”. In CAV 2002, vol. 2404 of LNCS, pp. 526-538.

[Hoa03] Tony Hoare, “The Verifying Compiler: A Grand Challenge for Computing Research”, Journal of the ACM, 50(1), January 2003, pp. 63–69.

[Hol03] G. J. Holzmann, “The SPIN Model Checker: Primer and Reference Manual”, Addison-Wesley, 2003.

[HS99] G. J. Holzmann and M.H. Smith, “Software Model Checking: Extracting verification models from source code”, *Formal Methods for Protocol Engineering and Distributed Systems,* (Conference Proceedings FORTE/PSTV99), Kluwer Academic Publ., Oct. 1999, pp. 481-497. See: http://cm.bell-labs.com/cm/cs/what/modex/

[JH07] Rajeev Joshi and Gerard J. Holzmann, “A Mini Challange: Build a Verifiable Filesystem”, in [MW07].

[Jon90] C. B. Jones, “Systematic Software Development using VDM”, Prentice-Hall, 1990

[LB98] B. Lewis and D. J. Berg, “Multithreaded programming with *Pthreads*”. Prentice-Hall, 1998.

[Mal06] Mahyar R. Malekpour, "A Byzantine Fault-Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems", NASA/TM-2006-214322

[Min93] Paul S. Miner. "Verification of Fault-Tolerant Clock Synchronization Systems”. NASA TP-3349, November, 1993.

[ML06] J. T. Mühlberg and G. Lüttgen, “Blasting linux code”. In FMICS 2006, LNCS 4346, pp. 211 226, 2006.

[MW07] B. Meyer and J. Woodcock (Eds), “Proceedings of Verified Software: Theories, Tools, Experiments (VSTTE) 06”, LNCS 4171, Springer, 2007.

[Ope03] The Open Group, The POSIX 1003.1, 2003 Edition Specification, available online at http://www.opengroup.org/certification/idx/posix.html.

[SC07] Radu Siminiceanu, Gianfranco Ciardo, "Formal verification of the NASA runway safety monitor”, *STTT* 9(1): p.63-76, 2007

[Spi95] M. Spivey, “The Z Reference Manual – 2nd edition”, Prentice-Hall, 1995.

[YCL07a] A. J. Yu, G. Ciardo, and G. Lüttgen, “Bounded reachability checking of asynchronous systems using decision diagrams”, In TACAS, vol. 4424 of LNCS, pp. 648–663. Springer, 2007.

[YCL07b] A. J. Yu, G. Ciardo, and G. Lüttgen, “Improving static variable orders via invariants”, In Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN), vol. 4546 of LNCS, pp. 83–103. Springer, 2007.

[YST+06] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, Dawson Engler, “Automatically Generating Malicious Disks using Symbolic Execution”, Proceedings of IEEE Security and Privacy, 2006

[YTME04] Junfeng Yang, Paul Twohey, Madanlal Musuvathi, Dawson Engler, “Using Model Checking to Find Serious File System Errors”, OSDI04, 2004

Appendices

Appendix A – Example of struct Definition

The following is part of the dcache.h file describing the dentry structure.

```
/*
 * linux/include/linux/dcache.h
 *
 * Dirent cache data structures
 *
 * (C) Copyright 1997 Thomas Schoebel-Theuer,
 * with heavy changes by Linus Torvalds
 */


struct dentry {
        atomic_t d_count;
        unsigned int d_flags;           /* protected by d_lock */
        spinlock_t d_lock;              /* per dentry lock */
        struct inode *d_inode;          /* Where the name belongs to - NULL is
                                         * negative */
        /*
         * The next three fields are touched by __d_lookup.  Place them here
         * so they all fit in a cache line.
         */
        struct hlist_node d_hash;       /* lookup hash list */
        struct dentry *d_parent;  /* parent directory */
        struct qstr d_name;

        struct list_head d_lru;         /* LRU list */
        /*
         * d_child and d_rcu can share memory
         */
        union {
                struct list_head d_child; /* child of parent list */
                struct rcu_head d_rcu;
        } d_u;
        struct list_head d_subdirs;     /* our children */
        struct list_head d_alias; /* inode alias list */
        unsigned long d_time;           /* used by d_revalidate */
        struct dentry_operations *d_op;
        struct super_block *d_sb;       /* The root of the dentry tree */
        void *d_fsdata;                 /* fs-specific data */
#ifdef CONFIG_PROFILING
        struct dcookie_struct *d_cookie; /* cookie, if any */
#endif
        int d_mounted;
        unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
};
```

Appendix B – Data Abstraction

B.1 Identifying Fields of Interest

Superblock

Attribute	Type	Description	Relevance	Rationale
s_list	struct list_head	Double link to other filesystems (superblocks)	Low	We'll only be using one file system
s_dev	dev_t	Device supporting filesystem	Low	We don't care about the device
s_blocksize	unsigned long	Size of file system	Low	We can fix this in our model
s_blocksize_bits	unsigned char	Size in bits of address space.	Low	We can fix this in our model
s_dirt	unsigned char	Superblock has changed since last write	Medium	May feature later when we look at device
s_maxbytes	unsigned long long	Maximum File Size	Low	Not interested in large files
s_type	struct file_system_type *	Type of FS (e.g. EXT2)	Low	Only interested in one file sys type
s_op	struct super_operations *	Pointer to ops structure (e.g. for EXT2)	Low	Only interested in one set of ops, characterised by our model
s_dq_op	struct duot_operations *	Disk quota ops	Low	Not interested in DQ
s_qcop	struct quotactl_ops *	Disk quota control ops	Low	Not interested in DQ
s_export_op	struct export_operations *	Export ops	Low	NFS only
s_flags	unsigned long	E.g. read only FS E.g. other inode flags	Low	Some flags might be important to correctness or fairness? Flags restrict full behaviour so can ignore (allow full functionality)
s_magic	unsigned long	Device ID number	Low	Not interested in device
s_root	struct dentry *	Dentry for root of FS	High	Important for correctness or consistency
s_umount	struct rw_semaphore	Reader/Writer semaphore (locks writes, permits reads) for use in unmounting	High	Will be important when modelling mount/unmount behaviour
s_lock	struct mutex	Superblock lock. Mutex structure. Modelling in previous versions as no of processes waiting and waiting queue	High	Correctness
s_count	int	Number of dentries referring to superblock	Medium	Consistency
s_syncing	int	Set during device synchronisation (writing changes)	Low	Not modelling device synchronisation yet.
s_need_synch_fs	int	Set when synchronisation required (similar to dirt)	Low	Not modelling device synchronisation yet.
s_active	atomic_t	Device Active?	Low	Not modelling device yet.
s_security	void *	Used in device security?	Low	Not modelling device or security properties
s_xattr	struct xattr_handler **	Concerning extended permissions	Low	Not modelling extended permissions
s_inodes	struct list_head	List of all Inodes	Medium	Ignore for now – can get to via Root
s_dirty	struct list_head	List of dirty inodes	Medium	Consistency. But not modelling device synch yet.
s_io	struct list_head	"Parked for writeback" list	Low	Ignoring
s_anon	struct hlist_head	Anonymous Dentries list	Low	Used by NFS
s_files	struct list_head	List of open files	Medium	Not modelling file info initially – may need abstract model of this eventually
s_bdev	struct block_device *	Device (backup?)	Low	Ignoring
s_instances	struct list_head	List of file system superblocks of same type	Low	Ignoring
s_quota_info	struct quota_info	Quota data	Low	No modelling quotas

s_frozen	int	Error handling – filesystem frozen	Low	Not modelling device/device failures
s_wait_unfrozen	wait_queue_head_t	Error handling – processes waiting for filesystem to be unfrozen	Low	Not modelling device/device failures
s_id	char _[32]	Name for information purposes	Low	Not important
s_fs_info	void *	Filesystem specific data	Low	For EXT2 this looks device specific
s_vfs_rename_mutex	struct mutex	Mutex used when renaming directories. Used by VFS.	Low	Not used by EXT2
s_time_gran	u32	Time granularity for atime/ctime etc.	Low	Not important to model. Hardwired.

Dentry

Attribute	Type	Description	Relevance	Rationale
d_count	atomic_t	Reference count - no of processes accessing dentry	High	Consistency and correctness
d_flags	unsigned int	Used by specific file sys implementations	Low	Never used by EXT2
d_lock	spinlock_t	Spinlock (used to protect dentry)	High	Correctness
d_inode	struct inode *	Pointer to Inode related to this dentry	High	Consistency/Correctness
d_hash	struct hlist_node	Links to other entries in hash bucket	Low	Not modelling dcache
d_parent	struct dentry *	Parent dentry or reflexive if root	High	Consistency
d_name	struct qstr	Name and hash value	Low	Use d_iname for name, simple hash
d_lru	struct list_head	Used to link unreferenced dentries (for mem management) See Note.	Medium	May cause problems. See note.
d_child	struct list_head	Used to link sibling Dentries [union with d_rcu]	High	Consistency
d_rcu	struct rcu_head	Queue of pending functions to be carried out on inode [union with d_child]	High	Correctness (but may have to limit queue size)
d_subdirs	struct list_head	Used to Link Child Dentries	High	Consistency
d_alias	struct list_head	Used to Link Dentries pointing to same inode (hard links)	Low	Ignoring Hard links for now
d_time	unsigned long	Used by FS implementations	Low	Used for shared file systems, therefore ignore for now
d_op	struct dentry_operations *	Pointer to operations on Dentries	Low	Captured by model
d_sb	struct super_block *	Pointer to superblock	Low	There will only be one initially
d_fsdata	void *	Used by FS implementations	Low	Not used by EXT2
d_cookie	struct dcookie_struct	Used in kernel analyses	Low	Not part of core implementation
d_mounted	int	Records whether Dentry carries a mount.	Low	Will only be mounting single file system, therefore no need for this.
d_iname	unsigned char _[16]	First 16 chars of name	High	Here's where we store our name.

Inode

Attribute	Type	Description	Relevance	Rationale
i_hash	struct hlist_node	Used to link inodes in this hash bucket	Low	No hashing needed on inodes
i_list	struct list_head	Links inodes in same state (used, unused,dirty) for entire FS	Medium	May be able to generate on-the-fly if important
i_sb_list	struct list_head	Used to link all Inodes (for superblock)	Low	Ignore (equivalents possible without allocating dataspace)
i_dentry	struct list_head	List of dentries referring to this inode	High	Consistency
i_ino	unsigned long	Inode identifier	Low	Inode identified by its “address” in pool

i_count	atomic_t	Number of processes	Medium	Consistency – but can ignore for now (no links, single mount)
i_mode	umode_t	File mode e.g permissions	Low	Only using one type of file
i_nlink	unsigned int	Number of hard links	Low	Not modelling links yet
i_uid/ i_gid	uid_t/gid_t	User/group ids	Low	Unimportant
i_rdev	dev_t	Device	Low	Device (for mouse etc.)
i_size	loff_t	File size	Low	Not modelling this (yet).
i_atime, i_mtime, i_ctime	struct timespec	Create, modify times etc.	Low	Unimportant.
i_blkbits	unsigned int	Blocksize (no of bits)	Low	Pertains to Device
i_blksize	unsigned long	Blocksize	Low	Pertains to Device
i_version	unsigned long	Used to track changes to inode (along with dirty)	Medium	Not modelling device yet
i_blocks	blkcnt_t	Filesize (blocks)	Low	Not modelling device yet
i_bytes	unsigned short	Filesize (bytes [in last block?])	Low	Not modelling device yet
i_lock	spinlock_t	Used to protect file size fields?	High	Correctness – but may be able to avoid if only used on these fields
i_mutex	struct mutex	Used to protect inode attributes	High	Correctness
i_alloc_sem	struct rw_semaphore	Used to protect inode attributes	High	Correctness
i_op	struct inode_operations *	Inode operations	Low	In model
i_fop	const struct file_operations *	File operations	Low	In model
i_sb	struct super_block *	Pointer to superblock	Low	Only one superblock
i_flock	struct file_lock *	File lock – See note	High	Correctness
i_mapping	struct address_space *	Mapping of inode to VM	Low	VM – avoid
i_data	struct address_space	More to do with VM	Low	VM – avoid
i_dquot	struct dquot * _ [X]	Disk quota info	Low	Not interested
i_devices	struct list_head	List of devices	Low	Device Specific
i_pipe	struct pipe_inode_info *	Pipe info	Low	Device Specific
i_bdev	struct block_device *	Block device info	Low	Device Specific
i_cdev	struct cdev *	Device info	Low	Device Specific
i_cindex	int	Device info	Low	Device Specific
i_generation	__u32	Used for security purposes	Low	Not interested in security
i_dnotify_mask	unsigned long	For directory notify	Low	Kernal option (trace/debug)
i_dnotify	struct dnotify_struct *	For directory notify	Low	Kernal option (trace/debug)
inotify_watches	struct list_head	Watches this inode	Low	Kernal option (trace debug?)
inotify_mutex	struct mutex	Used when watching inode	Low	Kernal option (trace debug?)
i_state	unsigned long	e.g. Dirty/locked/freeing	High	Correctness.
dirtied_when	unsigned long	Time stamp of first dirtying	Low	Not modelling dirtying yet.
i_flags	unsigned int	Type of inode (incl type of locking)	Low	Only modelling most liberal type.
i_writecount	atomic_t	Write access count/deny count (no of processes)	High	Consistency (with processes)
i_security	void *	Used for security purposes	Low	Not interested in security
generic_ip	void *	Don't know	Low	Not used in code
i_size_seqcount	seqcount_t	Kernal option	Low	Kernal option (trace?)

B.2. Information Modelling

Superblock

Attribute	Description	Relevance	Bit Estimates	Rationale
s_root	Dentry for root of FS	High	3	Based on max of 8 dentries
s_umount	Reader/writer semaphore	High	3	Provisionally modelled in same way as mutex structure.
s_lock	Superblock lock – Mutex Structure (count, spinlock, wait queue)	High	3 (1 bit lock, 1 bit PID of process holding lock, 1 bit PID of waiting process [if different from holding process])	Based on up to 1 waiting process – doesn't quite reflect mutex structure, but hopefully close enough. Assumes 2 processes.

Dentry

Attribute	Description	Relevance	Bit Estimates	Rationale
d_count	Reference count (no of children)	High	3	Allows for 6 processes
d_lock	Spinlock (used to protect dentry)	High	3	As for s_lock
d_inode	Pointer to inode related to this dentry	High	3	8 possible inodes
d_parent	Parent dentry or reflexive if root	High	3	7 possible parents (all dentries have parents, roots parent is itself)
d_child	List of siblings	High	8	Based on "marking" of relevant entries. Large model probably better served with constrained DMA.
d_rcu	Queue of pending functions to be carried out on inode [union with d_child]	High	4	This is a union with d_child (only applies to root?). However, we budget separately. 4 bits includes 1 bit to identify that an operation is pending and 3 to identify that operation. If more than 1 operation needed it will need to be a resource failure (successful termination). This can be reviewed in the light of further modelling.
d_subdirs	List of children	High	8	Based on marking of relevant entries. Large model probably better served with constrained DMA.
d_iname	First 16 chars of name	High	3	We need up to 8 "names"

Inode

Attribute	Description	Relevance	Bit Estimates	Rationale
i_dentry	List of dentries referring to this node	High	3	There's only going to be one (no links)
i_lock	Used to protect file size fields?	High	3	As for superblock lock
i_mutex	Used to protect inode attributes	High	3	As for superblock mutex
i_alloc_sem	Used to protect inode attributes	High	3	Modelling as lock for now
i_flock	File locks – see note	High	9	3 bits per queue (list of processes holding locks [0/1 length 2], list of type of lock [0/1, length 2], list of possible process blocks [0/1, length 2, 0 means no process blocked 1 means process not holding lock blocked])
i_state	e.g. Dirty/locked/freeing	High	2	Assume states of interest can be modelled in 2 bits
i_writecount	Write access count/deny count (no of processes with write access)	High	3	Either –1, 0, 1 or 2 (assuming max two processes)

Appendix C - Pseudocode

C.1 Miscellaneous Supporting Functions

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_misc.c,v $
 * $Revision: 1.16 $, $Date: 2007/08/14 16:26:35 $
 */

/* Defines: */

#define NULL    (void *) 0

#define DIRTY    1
#define DELETING 2
#define TRUE     1
#define FALSE    0

#define ERROR    0
#define SUCCESS  1

/* Types: */
typedef char * string; /* just in order to remove pointers */

typedef int spinlock_t;

typedef int atomic_t;

typedef int mutex_t;

typedef struct inode_t
 {
  int        is_allocated;
  atomic_t   i_count;
  spinlock_t i_lock;
  int        i_state;
  mutex_t    i_mutex;
 } inode_t;

inode_t iNULL = {0, 0, 0, 0};

typedef struct dentry_t
 {
  int        id; /* in order to avoid random bit shifting operations
                  * in the pseudocode I assume dentries to be numbered
                  * by 2^n with n being the decimal number of the entry. */
  int        is_allocated;
  void       *d_parent;
  atomic_t   d_count;
  spinlock_t d_lock;
  int        d_child;
  int        d_subdirs;
  inode_t    *d_inode;
  string     d_iname;
 } dentry_t;

dentry_t dNULL = {0, 0, NULL, 0, 0, 0, 0, NULL, NULL};

typedef struct lookup_res_t /* just in order to remove pointers */
 {
  dentry_t  * parent;
  dentry_t  * file;
 } lookup_res_t;

lookup_res_t lNULL = {&dNULL, &dNULL};
```

```
/* Function defs: */
#include "pseudo.h"


/* Hacks: */
static dentry_t *root    = NULL; /* this is a pointer to the root directory;
                                  * should be part of the superblock */
static dentry_t *current = NULL; /* we assume that we have a global pointer to
                                  * the current directory of each process */


/* Global locks: */
spinlock_t inode_lock;
spinlock_t dcache_lock;


/* General comments:
 * - Most of the functions defined here use local variables such as
 *   counters or temporary dentries. Of course this is not thread-save.
 *   A model implementing the functions should either inline them
 *   or use some additional locking around each function in order to
 *   serialise their execution.
 * - This pseudocode models a mixture between a synchronous and an
 *   asynchronous filesystem. Especially the way how we wait until
 *   nobody else uses a particular dentry or inode we want to delete,
 *   is significantly different from the Linux kernel's operation.
 *   The reason for this is that I didn't want to introduce a scheduler,
 *   software interrupts and additional process lists.
 * - There will be a new version of these code snippets containing
 *   cross-references to the kernel's code.
 * - The whole pseudocode is about 10 pages long now. Since I have
 *   no means of compiling or testing it I would not expect it to be
 *   free of errors :-)
 */


/* Functions not defined explicitly:
 * - find_dentry() -- a model specific function that returns a dentry
 *   for a given parent and a filename */
extern dentry_t find_dentry   (dentry_t, string);

/* - sleep() -- waits for some time. */
extern void sleep             (void);

/* - foreach(array) -- do something with every element of array */
extern void foreach           (string);
/* - cont() -- is the continue-statement to be used in the foreach-loop. */
extern void cont              (void);

/* - spin_lock(), spin_unlock() -- spinlock operations */
extern void spin_lock         (spinlock_t);
extern void spin_unlock       (spinlock_t);

/* - first_component(), next_component(), last_component() are
 *   functions "exploding" path strings into it's components separated
 *   by /. If / is the first component of a path, it refers to the
 *   root directory. All other /es handled as delimiters. */
extern string first_component (string);
extern string next_component  (string);
extern string last_component  (string);
extern string concat          (string, string);

/* - atomic_read() and atomic_write() read and write atomic integer
 *   variables. */
extern int  atomic_read       (atomic_t);
extern void atomic_write      (atomic_t, int);
extern void atomic_inc        (atomic_t);
extern void atomic_dec        (atomic_t);

/* - up() and down() set and anset mutexes; we can probably skip those. */
extern void up               (mutex_t);
extern void down             (mutex_t);

/* - allocate_inode(dentry) returns a new inode; i_count is set to 1,
 *   i_dentry points to dentry, all locks are released, state is dirty. */
extern inode_t allocate_inode (dentry_t);

/* - allocate_dentry(filename, parent) returns a new dentry; d_iname is
 *   set to filename, d_parent is set to parent, all locks are released,
 *   all lists are empty, d_count is 1. */
extern dentry_t allocate_dentry (string, dentry_t);

/* - deallocate_dentry() and deallocate_inode() set every data field
 *   in a given dentry or inode to 0. */
extern void deallocate_dentry (dentry_t);
extern void deallocate_inode  (inode_t);


/* Initialisation */

int init_fs (void)
 {
  dentry_t my_root;
  inode_t  itmp;

  spin_lock (dcache_lock);
  spin_lock (inode_lock);

  /* get dentry for / */
  my_root = allocate_dentry ("/", dNULL);
  if (!my_root.is_allocated) { goto FIN; }

  /* get inode for / */
  itmp = allocate_inode (my_root);
  if (!itmp.is_allocated) { goto FIN; }
  my_root.d_inode   = &itmp;      /* set inode */
  my_root.d_parent  = &my_root;   /* root is its own parent  */
  my_root.d_subdirs = my_root.id; /* root is its own subdir  */
  my_root.d_child   = my_root.id; /* root is its own sibling */

  /* set up "superblock" */
  root = &my_root;

FIN:
  spin_unlock (inode_lock);
  spin_unlock (dcache_lock);

/* /lost+found is optional */
#ifdef __HAVE_LOSTANDFOUND
  if (root && sys_mkdir ("/lost+found") == SUCCESS)
#else
  if (root)
#endif
   { return (SUCCESS); }
   else
   { return (ERROR); }
 }


/* Cleanup process: */

void cleanup (void)
 {
  inode_t  inode;
  dentry_t dentry;

  while (1) /* This should only be an endless loop if it's actually
             * running as a separate process. Otherwise it must
             * terminate in order to avoid deadlock situations. */
   {
    spin_lock (dcache_lock);
    foreach ("dentry in /list of dentries/");
     {
      if (! atomic_read(dentry.d_count))
       { deallocate_dentry (dentry); }
     }
    spin_unlock (dcache_lock);

    spin_lock (inode_lock);
    foreach ("inode in /list of inodes/");
     {
      if (! atomic_read(inode.i_count))
       { deallocate_inode (inode); }
      if (inode.i_state == DIRTY) /* sync operation */
       {
        spin_lock (inode.i_lock);
        inode.i_state = 0;
        spin_unlock (inode.i_lock);
       }
     }
    spin_unlock (inode_lock);

    sleep();
   }

  return;
 }


/* Helper functions: */

/* is a given dentry a directory? */
int is_directory (dentry_t dentry)
 {
  if (dentry.is_allocated && dentry.d_subdirs != 0)
   { return (TRUE); }

  return (FALSE);
 }

/* increment d_count */
void dget (dentry_t dentry)
 {
  if (dentry.is_allocated)
   {
    spin_lock (dentry.d_lock);
    if (atomic_read (dentry.d_count))
     { atomic_inc(dentry.d_count); }
    spin_unlock (dentry.d_lock);
   }

  return;
 }

/* decrement d_count */
void dput (dentry_t dentry)
 {
  if (dentry.is_allocated)
   {
    spin_lock (dentry.d_lock);
    if (atomic_read (dentry.d_count) > 1)
     { atomic_dec (dentry.d_count); }
    spin_unlock (dentry.d_lock);
   }

  return;
 }

/* returns dentry for parent/path if it exists, NULL otherwise */
dentry_t get_dentry (string path, dentry_t parent)
 {
  dentry_t dtmp;

  spin_lock (dcache_lock);
  dtmp = find_dentry (parent, path);  /* model specific function */
  if (dtmp.is_allocated)
   {
    dget (dtmp);                     /* mark entry as "in use" */
    if (!atomic_read (dtmp.d_count)) /* did it work? */
     { dtmp = dNULL; }               /* error */
   }
  spin_unlock (dcache_lock);

  return (dtmp);
 }

/* THIS IS THE NEW VERSION OF get_dentry() AS DISCUSSED WITH ANDY
 * VIA MAIL. */
/* returns dentry for parent/path if it exists, NULL otherwise */
dentry_t get_dentry_NEW (string path, dentry_t parent)
{
 dentry_t dent;

 spin_lock (dcache_lock);

 foreach ("dent in /list of dentries/");
  {
   if (!dent.is_allocated) { cont(); }  /* dent is not allocated */
   if (((dentry_t *)dent.d_parent)->id == parent.id &&  /* correct path and */
       dent.d_iname == path)          /* correct filename? */
    {
     dget (dent);                     /* mark entry as "in use" */
     if (!atomic_read (dent.d_count)) /* did it work? */
      { cont(); }                   /*  no. check next entry. */
      else
      { spin_unlock (dcache_lock);
        return (dent); }              /*  yes! return entry. */
    }
  }
 spin_unlock (dcache_lock);           /* no matching entry found. */

 return (dNULL);
}

/* path traversal, returns parent's and child's dentries */
lookup_res_t path_lookup (string path)
 {
  lookup_res_t result;
  dentry_t parent = dNULL, dtmp;
  string tmp;

  if (path[0] != '/')     /* if path does not start with / */
   {
    if (current != NULL)
     { parent = *current; } /* get current working dir */
     else
     { return (lNULL); } /* error */
   }

  tmp = first_component (path); /* this will return "/" */
  while (tmp)
   {
    /* get dentry for current path component */
    dtmp = get_dentry (tmp, parent);

    if (!dtmp.is_allocated) /* --------------- current path does not exist */
     {
      if (!parent.is_allocated)
       { return (lNULL); } /* error */
      if (tmp != last_component (path))
       { dput (parent); return (lNULL); } /* error */
       else
       { result.parent = &parent; result.file = &dNULL;
         return (result); }
     }

    /* -----------------------   current path does exist */
    if (tmp != last_component (path))
     {
      if (is_directory (dtmp))
       { /* continue path traversal */
        dput (parent); /* ! this line may be erroneous */
        parent = dtmp;
        tmp = next_component (path);
       }
       else
       { /* further traversal not possible because one middle
          * component is regular file */
        dput (parent);
        dput (dtmp);
        return (lNULL); /* error */
       }
     }
     else
     { /* this is the last component; we are done. */
      tmp = NULL;
     }
   } /* while (tmp) */

  result.parent = &parent; result.file = &dtmp;
  return (result);
 }

/* release usage counters for an open file */
void path_release (dentry_t dentry)
 {
  if (dentry.is_allocated)
   {
    dput (*((dentry_t *)(dentry.d_parent)));
    dput (dentry);
   }

  return;
 }

/* update sibling list for each child of parent;
 * update parent.d_subdirs */
void update_parent (dentry_t parent)
 {
  dentry_t dent;
  int siblings = 0, subdirs = 0;

  if (!parent.is_allocated) { return; }
  if (!is_directory(parent)) { return; }
  /* find subdirs and siblings */
  foreach ("dent in /list of dentries/");
   {
    if (dent.id != root->id && /* exclude root, in the models this should
                               * become something like (dent.id != 0) */
        ((dentry_t *) dent.d_parent)->id == parent.id &&
        atomic_read(dent.d_count) )
     { siblings |= dent.id;  /* refers to dent's ID */
       if (is_directory(dent))
        { subdirs |= dent.id; }
     }
   }

  /* update siblings */
  foreach ("dent in /list of dentries/");
   {
    if (dent.id != root->id && /* exclude root, in the models this should
                               * become something like (dent.id != 0) */
        ((dentry_t *) dent.d_parent)->id == parent.id &&
        atomic_read(dent.d_count) )
     { dent.d_child = siblings; }
   }

  /* update subdirs; make sure that / is a subdir of / */
  parent.d_subdirs = subdirs | parent.id;

  return;
 }
```

C.2 Creating a file

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_creat.c,v $
 * $Revision: 1.11 $, $Date: 2007/11/09 17:51:15 $
 */

/* sys_creat is actually a specific behaviour of sys_open() */
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);
 }
```

C.3 Deleting a File

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_unlink.c,v $
 * $Revision: 1.9 $, $Date: 2007/06/27 13:34:02 $
 */

int sys_unlink (string path)
 {
 lookup_res_t l;
 dentry_t parent, file;

 l = path_lookup (path);
 parent = *l.parent;
 file = *l.file;

 if (!file.is_allocated || is_directory (file))
  { dput (file); dput (parent); return (ERROR); }

 down (file.d_inode->i_mutex); /* to be cleared at the point of
                               * re-allocation! */

 spin_lock (dcache_lock); /* d_delete () */

 while (atomic_read(file.d_count) != 0) /* dentry_iput */
  {
   spin_lock (file.d_lock);
   if (atomic_read(file.d_count) == 2)
    { atomic_write(file.d_count, 0); }
   spin_unlock (file.d_lock);
  }

 /* there may be a bug in this line; I'm not sure when exactly
  * i_count is decremented or incremented. */
 spin_lock (inode_lock);
 while (atomic_read(file.d_inode->i_count) != 0) /* iput () */
  {
   spin_lock (file.d_inode->i_lock);
   if (atomic_read(file.d_inode->i_count) == 1)
    { atomic_write (file.d_inode->i_count, 0); }
   spin_unlock (file.d_inode->i_lock);
  }
 spin_unlock (inode_lock);

 update_parent (*((dentry_t *)file.d_parent));
 dput (parent);
 spin_unlock (dcache_lock);

 return (SUCCESS);
 }
```

C.4 Creating a Directory

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_mkdir.c,v $
 * $Revision: 1.10 $, $Date: 2007/08/13 17:22:05 $
 */

int sys_mkdir (string path)
 {
  lookup_res_t l;
  inode_t itmp;
  dentry_t parent, dir;

  l = path_lookup(path);
  parent = *l.parent;
  dir = *l.file;

  if (dir.is_allocated)
   { path_release (dir); return (ERROR); }
  if (!parent.is_allocated)
   {
    if (dir.is_allocated) /* deals with root look up */
     { dput(dir); }
    return (ERROR);
   }

  spin_lock (dcache_lock);
  dir = allocate_dentry(last_component(path), parent);
  if (!dir.is_allocated)
   { spin_unlock (dcache_lock);
     dput(parent);
     return (ERROR); }

  dget(dir);

  spin_lock (inode_lock);
  itmp = allocate_inode(dir);
  dir.d_inode = &itmp;
  if (dir.d_inode->is_allocated) { down(dir.d_inode->i_mutex); }
  spin_unlock (inode_lock);

  if (!dir.d_inode->is_allocated)
   { atomic_write (dir.d_count, 0);
     dput (parent);
     spin_unlock (dcache_lock);
     return (ERROR); }

  dir.d_subdirs  = dir.id;
  update_parent (*((dentry_t *)dir.d_parent));
  path_release (dir);
  spin_unlock (dcache_lock);

  up(dir.d_inode->i_mutex);

  return (SUCCESS);
 }
```

C.5 Removing a Directory

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_rmdir.c,v $
 * $Revision: 1.14 $, $Date: 2007/08/13 17:22:05 $
 */

int sys_rmdir (string path)
 {
  lookup_res_t l;
  dentry_t parent, dir, tmp;
  int children = 0;

  l = path_lookup (path);
  parent = *l.parent;
  dir = *l.file;

  /* 1. -- sanity checks */
  if (!dir.is_allocated || !is_directory(dir) || !parent.is_allocated)
   { dput (dir); dput (parent); return (ERROR); }

  /* 2. -- lock the node */
  down (dir.d_inode->i_mutex);
  spin_lock (dcache_lock); /* d_delete () */

  /* 3. -- check for subdirectories
   * (needs i_mutex and dcache_lock in order to avoid others changing
   * the state) */
  foreach ("tmp in /list of dentries/");
   {
    if (((dentry_t *)(tmp.d_parent))->id == dir.id &&
        atomic_read (tmp.d_count))
     { children++; }
   }

  if (children != 0)
   {
    spin_unlock (dcache_lock);
    up (dir.d_inode->i_mutex);
    dput (dir); dput (parent); /* no path_release because another process could
                               * have already destroyed the parent relation */
    return (ERROR);
   }

  /* 4. -- mark the node for deletion
   * (uses d_inode->i_mutex and d_inode->i_state) */
  if (dir.d_inode->i_state & DELETING)
   { /* somebody else is already deleting this node -- success */
    dput (dir); dput (parent); /* no path_release because another process
                               * has already destroyed the parent relation */
    up (dir.d_inode->i_mutex);
    return (SUCCESS);
   }
   else
   { /* we are going to delete this node */
    dir.d_inode->i_state |= DELETING;
    up (dir.d_inode->i_mutex);
   }

  /* ==> dir.d_inode->i_mutex is free now */

  /* 5. -- now remove links to this node so that later path_lookup()s won't
   * return it and we don't get any new processes working on this node */
  dir.d_parent = &dNULL;
  update_parent (parent); /* This can result in clients unsing invalid
                           * working directories. This is okay. */
  dput (parent);
  spin_unlock (dcache_lock);

  /* ==> dcache_lock is free now */
  /* ==> other processes may continue working on this directory here */

  /* 6. -- set dir.d_count to 0 */
  while (atomic_read(dir.d_count) != 0) /* dentry_iput */
   {
    spin_lock (dcache_lock);
    spin_lock (dir.d_lock);
    if (atomic_read(dir.d_count) == 2)
     { atomic_write(dir.d_count, 0); }
    spin_unlock (dir.d_lock);
    spin_unlock (dcache_lock);
   }

  /* 7. -- set dir.d_inode->i_count to 0 */
  /* ! there may be a bug in this line; I'm not sure when exactly
   * i_count is decremented or incremented. */
  spin_lock (inode_lock);
  while (atomic_read(dir.d_inode->i_count) != 1) /* iput () */
   {
    spin_lock (dir.d_inode->i_lock);
    if (atomic_read(dir.d_inode->i_count) == 1)
     { atomic_write (dir.d_inode->i_count, 0); }
    spin_unlock (dir.d_inode->i_lock);
   }
  spin_unlock (inode_lock);


  return (SUCCESS);
 }
```

C.6 Renaming a File or Directory

```
/*
 * $Author: muehlber $ : $RCSfile: pseudo_rename.c,v $
 * $Revision: 1.9 $, $Date: 2007/08/03 13:43:06 $
 */

int sys_rename (string src, string dst)
 {
 lookup_res_t l;
 dentry_t src_parent, src_file;
 dentry_t dst_parent = dNULL, dst_file = dNULL;

 l = path_lookup (src);
 src_parent = *l.parent;
 src_file = *l.file;

 if (!src_file.is_allocated)
  { goto MVERROR; }

 l = path_lookup (dst);
```

```
dst_parent = *l.parent;
dst_file = *l.file;
if (!dst_parent.is_allocated || src_file.id == dst_file.id)
 { goto MVERROR; }

if (is_directory(dst_file)) /* target is directory; move file into it */
 { dput (dst_parent);
   dst_parent = dst_file;
   dst_file = get_dentry(last_component(src), dst_parent);
   /* the implementation uses a temporary dentry; but since we
    * probably don't care about filenames at all, I use a string
    * operation. */
   dst = concat (dst, last_component(src)); }

if (is_directory(dst_file))
 { goto MVERROR; } /* EFAULT */

if (!is_directory(dst_file) && is_directory(src_file))
 { goto MVERROR; } /* EFAULT */

if (is_directory(src_file) && atomic_read(src_file.d_count > 2))
 { goto MVERROR; } /* EBUSY */

/* the implementation follows dst_file.d_parent.d_parent... and
 * checks whether there is a parent that equals src_file */
if ("the new pathname contained a path prefix of the  old")
 { goto MVERROR; } /* EINVAL; */

down (dst_parent.d_inode->i_mutex);

spin_lock (dcache_lock);

if (dst_file.is_allocated) /* remove dst_file */
 {
  while (atomic_read(dst_file.d_count) != 0) /* dentry_iput */
   {
    spin_lock (dst_file.d_lock);
    if (atomic_read(dst_file.d_count) == 2)
     { atomic_write(dst_file.d_count, 0); }
    spin_unlock (dst_file.d_lock);
   }

   /* there may be a bug in this line; I'm not sure when exactly
    * i_count is decremented or incremented. */
   spin_lock (inode_lock);
   while (atomic_read(dst_file.d_inode->i_count) != 0) /* iput () */
    {
     spin_lock (dst_file.d_inode->i_lock);
     if (atomic_read(dst_file.d_inode->i_count) == 1)
      { atomic_write (dst_file.d_inode->i_count, 0); }
     spin_unlock (dst_file.d_inode->i_lock);
    }
   spin_unlock (inode_lock);

  update_parent (*((dentry_t *)dst_file.d_parent));
  dst_file = dNULL; /* we are done with it */
 }

/* rename */
src_file.d_parent = &dst_parent;
src_file.d_iname = last_component (dst);
update_parent (src_parent);
update_parent (dst_parent);
dput (dst_parent);
dput (src_parent);
dput (src_file);
```

```
  spin_unlock (dcache_lock);

  up (dst_parent.d_inode->i_mutex);

  return (SUCCESS);

MVERROR:
  dput (dst_file); dput (dst_parent);
  dput (src_file); dput (src_parent);
  return (ERROR);
 }
```

Appendix D – Examples from the SPIN Model

D.1 Data Structures in Promela

```
typedef dentry {
        unsigned d_count : 3;
        lock d_lock;
        unsigned d_inode : 3;
        unsigned d_parent : 3;
        bit d_child [8];
        rcu d_rcu;
        bit d_subdirs [8];
        unsigned d_iname : 3
};

typedef inode {
        unsigned i_dentry : 3;
        lock i_lock;
        lock i_mutex;
        lock i_alloc_sem;
        filelock i_flock;
        unsigned i_state : 2;
        unsigned i_writecount : 3
};

typedef dentrypool {
        dentry dentries [NoofNodes];
        bit available [8]
};

typedef inodepool {
        inode inodes [NoofNodes];
        bit available [8]
};
```

D.2. Allocating and Deallocating Nodes

```
inline alloc_dentry (dep,returnval, localvar,error) {
d_step{
        localvar=0;
        do
        :: localvar==NoofNodes -> break
        :: else {
                if
                :: dep.available[localvar] != 0  -> localvar++
                :: else  {
                        dep.available[localvar]=1;
                        returnval=localvar;
                        break
                }
                fi
        }
        od;
        if
        :: localvar==NoofNodes -> error=1
        :: else error=0
        fi
} /*dstep */
        if
        :: error==1 ->
                goto end
        :: else
        fi
};

inline dealloc_dentry(dep,dent) {
d_step{
        assert (dent>=0 && dent<=NoofNodes-1 && dep.available[dent]==1);
        dep.available[dent]=0
} /*dstep*/
};
```

D.3 Other core functions

```
inline allocate_dentry(dent,name,parent,lvplus1_4h,lvplus1_1)
{
        alloc_dentry(dpool,dent,lvplus1_4h,lvplus1_1);
        /* init dentry defaults + parent + filename*/
        dpool.dentries[dent].d_count=1; /* Not marked for deletion */
        dpool.dentries[dent].d_lock.islocked=0;
        dpool.dentries[dent].d_lock.lockedby=0;
        dpool.dentries[dent].d_lock.waiting=0;
        dpool.dentries[dent].d_inode=0; /* initially 0 */
        dpool.dentries[dent].d_parent=parent; /* parent of root is root */
        dpool.dentries[dent].d_child[0]=0;
        dpool.dentries[dent].d_child[1]=0;
        dpool.dentries[dent].d_child[2]=0;
        dpool.dentries[dent].d_child[3]=0;
        dpool.dentries[dent].d_child[4]=0;
        dpool.dentries[dent].d_child[5]=0;
        dpool.dentries[dent].d_child[6]=0;
        dpool.dentries[dent].d_child[7]=0; /* No siblings */
        /* Not bothering to set rcu for now */
        dpool.dentries[dent].d_subdirs[0]=0;
        dpool.dentries[dent].d_subdirs[1]=0;
        dpool.dentries[dent].d_subdirs[2]=0;
        dpool.dentries[dent].d_subdirs[3]=0;
        dpool.dentries[dent].d_subdirs[4]=0;
        dpool.dentries[dent].d_subdirs[5]=0;
        dpool.dentries[dent].d_subdirs[6]=0;
        dpool.dentries[dent].d_subdirs[7]=0; /* No children (dirs) */
        dpool.dentries[dent].d_iname=name
}

inline modelfinddentry(name,parent,returndent,count)
{
        /* Assume dcache locked when called */
        assert(dcache_lock.islocked==1);
        if
        :: parent==NoofNodes -> { /*Parent Null*/
                assert(name==0); /* We should be looking for root */
                returndent=super.s_root /* And here's the root dentry */
        }
        :: else {
                /* This is the equivalent of the dcache operation to find the right name */
                /* Since speed is not important for model, will search dpool rather than use subdirs */
                count=0;
                do
                :: count<NoofNodes -> {
                        if
                        :: dpool.available[count]==1 && dpool.dentries[count].d_parent==parent &&
                                dpool.dentries[count].d_iname==name  &&
                                dpool.dentries[count].d_count!=0 -> { /* in use */
                                        returndent=count;
                                        break
                        }
                        :: else count++
                        fi
                }
                :: count==NoofNodes -> {
                        returndent=NoofNodes;
                        break
                }
                od
        }
        fi
}
```

D.4. Supporting Functions

```
inline get_dentry(name,parent,returndent,mfdlv_4_1) {
        spinlock_lock(dcache_lock);
        modelfinddentry(name,parent,returndent,mfdlv_4_1); /* specific find function */
        if
        :: returndent!=NoofNodes -> {  /* Found name */
                dget(returndent);
                assert(dpool.dentries[returndent].d_count != 0) /* replaces check for success in
pseudocode */
        }
        :: else
        fi;
        spinlock_unlock(dcache_lock)
}

inline update_parent(dent,siblingslv,subdirslv,isdirectory_flag,count,count2)
{
        assert(dent!=NoofNodes);
        printf("Update parent called with dent = %u\n",dent);
        is_directory(dent,isdirectory_flag);
        assert(isdirectory_flag==1);

        count=0;
        do /* init siblings and subdirs */
        :: count<NoofNodes -> {
                siblingslv[count]=0;
                subdirslv[count]=0;
                count++;
        }
        :: count==NoofNodes -> break
        od;

        count=1; /* Don't take root into account */
        do /* calc subdirs and siblings */
        :: count<NoofNodes -> {
                if
                :: dpool.available[count]==1 && dpool.dentries[count].d_count!=0 /* in use */
                        && dpool.dentries[count].d_parent==dent -> { /* and parent is right one */
                        siblingslv[count]=1; /* mark as sibling */
                        is_directory(count,isdirectory_flag);
                        if
                        :: isdirectory_flag -> subdirslv[count]=1; /* mark as subdir */
                        :: else
                        fi;
                        count++
                }
                :: else count++
                fi;
        }
        :: count==NoofNodes -> break
        od;
        subdirslv[dent]=1; /* dent is subdir of itself */

        count=1; /* Don't update root's siblings */
        do /* update siblings */
        :: count<NoofNodes -> {
                if
                :: dpool.available[count]==1 && dpool.dentries[count].d_count!=0 /* in use */
                        && dpool.dentries[count].d_parent==dent -> { /* and parent is right one */
                        count2=0;
                        do /* write sibling array */
                        :: count2<NoofNodes -> {
                                dpool.dentries[count].d_child[count2]=siblingslv[count2];
                                count2++
                        }
                        :: count2==NoofNodes -> break
                        od
                }
                :: else
                fi;
                count++;
        }
        :: count==NoofNodes -> break
        od;

        count2=0;
        do /* write subdirs array */
        :: count2<NoofNodes -> {
                dpool.dentries[dent].d_subdirs[count2]=subdirslv[count2];
                count2++
        }
        :: count2==NoofNodes -> break
        od
}

inline path_lookup(patharray,cwd,parent,child,tmp,dtmp,pathindex,isdirectory_flag,mfdlv_4_1) {
        parent=NoofNodes; /* Null */
        if
        :: patharray[0]!=0 -> prepend(patharray,cwd)
        :: else
        fi;
        assert(patharray[0]==0); /*starts with root */
        tmp=0; /* first name is root */
        pathindex=0; /* current position in path */
        do
        :: tmp != NoofNodes -> { /* Not reached end of path (NoofNodes=NULL) */
                printf("tmp=%u, pathindex=%u\n",tmp,pathindex);
                get_dentry(tmp,parent,dtmp,mfdlv_4_1);
                printf("dtmp fetched as %u\n",dtmp);
                if
                :: dtmp==NoofNodes -> { /* current path extension doesn't exist */
                        if
                        :: parent==NoofNodes -> {/* Never even found root */
                                child=NoofNodes;
                                break
                        }
                        :: else { /* Found root but got stuck later */
                                if
                                :: patharray[pathindex+1]!=NoofNodes -> { /* Got stuck before last
component */
                                        dput(parent);
                                        parent=NoofNodes;
                                        child=NoofNodes;
                                        break
                                }
                                :: else { /* It was the last component in the path */
                                        child=NoofNodes;
                                        printf("Last component in Path!!\n");
                                        break
                                }
                                fi
                        }
                        fi
                }
                :: else  { /* Current path extension does exist */
                        if
                        :: patharray[pathindex+1]!=NoofNodes -> { /* Not last element in path */
                                is_directory(dtmp,isdirectory_flag);
                                if
                                :: isdirectory_flag -> {
                                        if
                                        :: parent!=NoofNodes -> dput(parent)
                                        /* First time through parent = NULL, no dget on parent, so
no dput */
                                        :: else
                                        fi;
                                        parent=dtmp;
                                        pathindex++;
                                        tmp=patharray[pathindex]
                                }
                                :: else { /* can't go any further */
                                        if
                                        :: parent!=NoofNodes -> dput(parent)
                                        /* First time through parent = NULL, no dget on parent, so
no dput */
                                        :: else
                                        fi;
                                        dput(dtmp);
                                        parent=NoofNodes;
                                        child=NoofNodes;
                                        break;
                                }
                                fi
                        }
                        :: else { /* Is last element */
                                pathindex++;
                                tmp=patharray[pathindex]
                        }
                        fi
                }
                fi
        }
        :: tmp == NoofNodes -> {
                child = dtmp;
                break /* parent and child set correctly */
        }
        od
}
```

D.5 Creating a file

```
inline
sys_creat(patharray,cwd,error,parent,file,isdirectory_flag,filename,plulv_4_1,plulv_4_2,plulv_4_3,plulv_1
_1,
                        up_lv1,up_lv2,mfdlv_4_1){
        error=0;
        path_lookup(patharray,cwd,parent,file,plulv_4_1,plulv_4_2,plulv_4_3,plulv_1_1,mfdlv_4_1);
        if
        :: parent!=NoofNodes -> { /* if parent exists */
                down(ipool.inodes[dpool.dentries[parent].d_inode].i_mutex);
                if
                :: file!=NoofNodes -> is_directory(file,isdirectory_flag) /* Only check dir if file exists */
                :: else
                fi;
                if
                :: file != NoofNodes && ! isdirectory_flag -> { /* File exists and isn't directory */
                        up(ipool.inodes[dpool.dentries[parent].d_inode].i_mutex);
                        path_release(file)
                }
                :: file != NoofNodes && isdirectory_flag -> { /* File exists but is directory */
                        up(ipool.inodes[dpool.dentries[parent].d_inode].i_mutex);
                        path_release(file);
                        error=1
                }
                :: file == NoofNodes -> { /* File doesn't exist */
                        spinlock_lock(dcache_lock);
                        last_component(patharray,filename); /* Get filename */
```

```
            allocate_dentry(file,filename,parent,plulv_4_1,plulv_1_1); /* borrow the PLU
locals! */
            assert(file!=NoofNodes && file!=0); /* replaces if structure - file not null or
root */
                    /* This check is redundant - root will never be reallocated
                     and allocate_dentry will terminate rather than return null
*/
            dget(file);
            spinlock_lock(inode_lock);
            allocate_inode(dpool.dentries[file].d_inode,file,plulv_4_1,plulv_1_1);
                        /* Borrowing pathlookup lvs again! */
            spinlock_unlock(inode_lock);
            assert(dpool.dentries[file].d_inode!=NoofNodes &&
                dpool.dentries[file].d_inode!=0); /* replaces if structure */
                    /* This check is redundant - root will never be reallocated
                     and allocate_dentry will terminate rather than return null
*/
            update_parent(dpool.dentries[file].d_parent,up_lv1,up_lv2,plulv_1_1,
                plulv_4_1,plulv_4_2); /* Borrow the plu lvs */
            printf("Updated Parent!! \n");
            path_release(file);
            spinlock_unlock(dcache_lock);
            up(ipool.inodes[dpool.dentries[parent].d_inode].i_mutex);
        }
        fi
    }
    :: else { /* Either child and parent don't exist, or child is root (from path_lookup(root)) */
        error=1;
        if
        :: file!=NoofNodes -> {
            assert(file==0); /* can only happen for root */
            dput(file)
        }
        :: else
        fi
    }
    fi
}
```

D.6 The Test Harness body

```
active proctype test () {

    unsigned node : 3;
    bit onebitlv;
    bit flag,errorflag;

    byte srcpath [PathLength];
    byte dstpath [PathLength];
    byte tmppath [PathLength];
    byte cwd[PathLength];
    unsigned fourbitlv3 : 4, fourbitlv4: 4, fourbitlv5:4, fourbitlv6:4;
    unsigned fourbitlv7:4,fourbitlv8 : 4, fourbitlv9:4, fourbitlv10:4;
    unsigned threebitlv : 3;
    bit bitlv,bitlv2;
    bit bitarraylv_1 [NoofNodes];
    bit bitarraylv_2 [NoofNodes];

    /* Initialise Superblock */
    init_superblock(dpool,ipool,super,node,fourbitlv,onebitlv);

    cd(dstpath,0); /* dstpath=root */
    cd(srcpath,0); /* srcpath=root */
    cd(cwd,0);/* Set cwd to root - cwd never used by harness, all calls by abs path, this is just for
call i/f */
```

```
        do
        :: {
                printf("Choosing Id (1-NoofNodes-1)\n");
                choose_id(node); /* Set srcpath */
                printf("Choosing src cd (0:root,1:down,2:up,3:skip)\n");
                printf("Current src path=");
                print_path(srcpath);
#if !defined(myverif)
                STDIN?c;
#endif
                if
#if !defined(myverif)
                :: c==48 -> cd(srcpath,0) /* root */
#endif
                ::
#if !defined(myverif)
                c==49 ->
#endif
                        cd(srcpath,node) /* down to id */
                ::
#if !defined(myverif)
                c==50 ->
#endif
                        cd(srcpath,NoofNodes) /* .. */
                ::
#if !defined(myverif)
                c==51 ->
#endif
                        skip
                fi;
#if !defined(myverif)
                STDIN?c; /* carriage return */
#endif

                printf("Choosing Id (1-NoofNodes-1)\n");
                choose_id(node); /* Set dstpath */
                printf("Choosing dst cd (0:root,1:down,2:up,3:skip)\n");
                printf("Current dst path=");
                print_path(dstpath);
#if !defined(myverif)
                STDIN?c;
#endif
                if
#if !defined(myverif)
                :: c==48 -> cd(dstpath,0)
#endif
                ::
#if !defined(myverif)
                c==49 ->
#endif
                        cd(dstpath,node)
                ::
#if !defined(myverif)
                c==50 ->
#endif
                        cd(dstpath,NoofNodes)
                ::
#if !defined(myverif)
                c==51 ->
#endif
                        skip
                fi;
#if !defined(myverif)
                STDIN?c; /* carriage return */
#endif
```

```
			printf("Choosing model functions
(0:mkdir,1:creat,2:rename,3:unlink,4:rmdir,5:skip)\n");
#if !defined(myverif)
			STDIN?c;
#endif
			if  /* Possibly call one of the Model functions */
			::
#if !defined(myverif)
			c==48 ->
#endif
			{
				sys_mkdir(srcpath,cwd,errorflag,fourbitlv3,fourbitlv4,bitlv,threebitlv,

		fourbitlv5,fourbitlv6,fourbitlv7,bitlv2,bitarraylv_1,bitarraylv_2,fourbitlv8);
				printf("Called Mkdir. Error=%u. Path=",errorflag);
				print_path(srcpath)
			}
			::
#if !defined(myverif)
			c==49 ->
#endif
			{
				sys_creat(srcpath,cwd,errorflag,fourbitlv3,fourbitlv4,bitlv,threebitlv,

		fourbitlv5,fourbitlv6,fourbitlv7,bitlv2,bitarraylv_1,bitarraylv_2,fourbitlv8);
				printf("Called Creat. Error=%u. Path=",errorflag);
				print_path(srcpath)
			}
			::
#if !defined(myverif)
			c==50 ->
#endif
			{
				copy_path(dstpath,tmppath);

		sys_rename(srcpath,tmppath,cwd,errorflag,fourbitlv3,fourbitlv4,fourbitlv9,fourbitlv10,

		bitlv,fourbitlv5,fourbitlv6,fourbitlv7,bitlv2,bitarraylv_1,bitarraylv_2,fourbitlv8);
					/* may alter paths!!!!!!!!!!!! */
				printf("Called Rename. Error=%u. Src Path=",errorflag);
				print_path(srcpath);
				printf("Dst Path=");
				print_path(dstpath)
			}
			::
#if !defined(myverif)
			c==51 ->
#endif
			{

		sys_unlink(srcpath,cwd,errorflag,fourbitlv3,fourbitlv4,bitlv,fourbitlv5,fourbitlv6,fourbitlv7,
					bitlv2,bitarraylv_1,bitarraylv_2,fourbitlv8);
				printf("Called Unlink. Error=%u. Path=",errorflag);
				print_path(srcpath)
			}
			::
#if !defined(myverif)
			c==52 ->
#endif
			{

		sys_rmdir(srcpath,cwd,errorflag,fourbitlv3,fourbitlv4,bitlv,fourbitlv5,fourbitlv6,fourbitlv7,
					bitlv2,bitarraylv_1,bitarraylv_2,fourbitlv8);
				printf("Called Rmdir. Error=%u. Path=",errorflag);
				print_path(srcpath)
			}
			::
```

```
#if !defined(myverif)
                    c==53 ->
#endif
                    {
                              skip;
                              printf("Called no function\n")
                    }
                    fi;
                    printf("\n");
#if !defined(myverif)
                    STDIN?c; /* carriage return */
#endif

progress_testharn:
                    printf("Report? (0:yes,1:no)\n");
#if !defined(myverif)
                    STDIN?c;
#endif
                    if /* Possibly report state of file system */
                    ::
#if !defined(myverif)
                    c==48 ->
#endif
                    {
                              printdentries(dpool,fourbitlv,fourbitlv2);
                              printf("\n\n");
                              printinodes(ipool,fourbitlv);
                    }
                    ::
#if !defined(myverif)
                    c==49 ->
#endif
                              skip
                    fi
#if !defined(myverif)
                    ;
                    STDIN?c /* carriage return */
#endif
          }
          od;
end:
          skip
}
```

D.7 Functions Supporting the Test Harness

```
inline cd(array,arg) /* scratch var cd_count */
{
          d_step {
                    if
                    :: arg==0 -> { /* cd root */
                              array[0]=0;
                              array[1]=NoofNodes
                    }
                    :: arg==NoofNodes -> { /* cd .. */
/*                            assert(array[0]==0 && array[1]!=NoofNodes);   Not root */
                              if /* replaces assertion for verification purposes */
                              :: array[0]!=0 || array[1]==NoofNodes
                              :: else {
                                        cd_count=0;
                                        do
                                        :: array[cd_count]!=NoofNodes -> cd_count++
                                        :: array[cd_count]==NoofNodes -> break
                                        od;
/*                                      assert(cd_count!=0); */
                                        if /* replaces assertion for verification purposes */
```

```
                :: cd_count==0
                :: else
                    array[cd_count-1]=NoofNodes
                fi
            }
            fi
        }
        :: else { /* cd id */
            cd_count=0;
            do
            :: array[cd_count]!=NoofNodes -> cd_count++
            :: array[cd_count]==NoofNodes -> break
            od;
/*          assert(cd_count+1<NoofNodes); */
            if /* replaces assertion for verification purposes */
            :: cd_count+1>=NoofNodes
            :: else
                concat_element(array,arg)
            fi
        }
        fi
    }
};


/* Choose an id between 1 and NoofNodes-1 - assumes max noofnodes is 8 */
inline choose_id (returnval)
{
    d_step{
#if !defined(myverif)
        STDIN?c;
#endif
        if
        ::
 #if !defined(myverif)
        c==49 &&
#endif
            NoofNames-1>=1 -> returnval=1
        ::
#if !defined(myverif)
        c==50 &&
#endif
            NoofNames-1>=2 -> returnval=2
        ::
#if !defined(myverif)
        c==51 &&
#endif
            NoofNames-1>=3 -> returnval=3
        ::
#if !defined(myverif)
        c==52 &&
#endif
            NoofNames-1>=4 -> returnval=4
        ::
#if !defined(myverif)
        c==53 &&
#endif
            NoofNames-1>=5 -> returnval=5
        ::
#if !defined(myverif)
        c==54 &&
#endif
            NoofNames-1>=6 -> returnval=6
        ::
#if !defined(myverif)
        c==55 &&
#endif
```

```
                                NoofNames-1>=7 -> returnval=7
                        fi
#if !defined(myverif)
                        ;
                        printf("c=%c, returnval=%u\n",c,returnval);
                        STDIN?c /* Get rid of carriage return */
#endif
            }
};

inline printdentries(dep,localvar,localvar2) { /* Needs 4 bit localvar */
d_step{
            localvar=0;
            printf("Dentry pool: \n\n");
            do
            :: localvar<=NoofNodes-1 -> {
                        if
                        :: dep.available[localvar] ==1 -> {
                                    printf("\t Dentry %u in use\n",localvar);
                                    printf("\t d_name = %u, d_inode = %u, d_parent = %u
\n",dep.dentries[localvar].d_iname,dep.dentries[localvar].d_inode,dep.dentries[localvar].d_parent);
                                    printf("\n");
                                    printf("d_count = %u\n\n", dep.dentries[localvar].d_count);
#if defined(myverif)
                                    assert(dep.dentries[localvar].d_count==1);
#endif
                                    printf("\t Siblings: ");
                                    print_relations(dep.dentries[localvar].d_child,localvar2);
                                    printf("\n \t Sub Directories: ");
                                    print_relations(dep.dentries[localvar].d_subdirs,localvar2);
                                    printf("\n\n");
                        }
                        :: else printf("\t Dentry %u not in use\n",localvar)
                        fi;
                        localvar++
                        }
            :: localvar==NoofNodes -> break
            od
} /*dstep */
};
```

Appendix E – Example from SMART Model

```
/*******************************************************
Abstract model of a virtual file system (originally EXT2)
Author: Radu Siminiceanu (NIA)
Lsst update: July 2, 2007
*******************************************************/

/* constants */

//int ND := 4;        /* maximum number of dentries */
//int NI := 4;        /* maximum number of inodes */
//int NP := 1;        /* maximum number of processes */

/* reserved file indices */
int ROOT       := 1;
int LOST_FOUND := 2;

/* i_node states */
int DELETING   := 1;

/* hash function */
/* does nothing for now */
/* could be used IF implementing the dcache */
int hashvalue(int x) := x;

/*=========================================*/
/* SMART options for state space generation */
/*=========================================*/
# Verbose true
# Report true                       /* reports MDD stats */
# IgnoreWeightClasses true          /* deals with immediate events priorities */
# Generations 5000                  /* For variable reordering, if needed  */
# GarbageSize 499000
# GarbageCollection GLOBAL

/* Recommended options for the state-space construction algorithm */
/* - Kronecker consistent: */
/*    MDD_SATURATION    -- standard on-the-fly: full MDD nodes   */
/*    MDD_SPARSE        -- on-the-fly: sparse MDD nodes          */
# StateStorage MDD_SPARSE
/* Non-KC algorithms are junk */

/*=========================================*/

spn EXT2(int nd, int ni, int np) := {

  /* Superblock */

  place
    superblock_root,                /* not used */
    superblock_umount_lock,         /* not used */
    superblock_lock,                /* not used */

    dcache_lock,             /* global lock on dcache */
    inode_lock;                     /* global lock on inodes */

    /* Convention used in this petri net:
       Locks and mutexes have the following values:
         if available,  >0
         not avalibale, =0
       I.e.: getting a lock/mutex removes a token
            releasing a lock/mutex adds the token back
    */

    init(dcache_lock:1, inode_lock:1);
    partition(3*nd-2+ni+8*np+1:dcache_lock:inode_lock);

  /* D_Entries */

  for (int i in {1..nd}) {
    place
      d_allocated[i],   /* is allocated? flag */
      d_parent[i],      /* id of parent: 0=n/a, or 1..ND */
      d_count[i],       /* reference count */
      d_lock[i],        /* not used */
      d_inode[i],       /* id of corresponding inode: 0=n/a, or 1..NI */
      d_subdirs[i];     /* number of subdirectories */

    /* put all places of dentry #i in partition #i */
    /* will result in large local subspace            */
    partition(
      cond(i>1,3*i-4,1):d_allocated[i]:d_subdirs[i]:d_count[i]:d_lock[i],
      cond(i>1,3*i-3,1):d_parent[i],
      cond(i>1,3*i-2,1):d_inode[i]
    );

    init(d_lock[i]:1);

  }

  /* Inodes section */

  for (int i in {1..ni}) {
    place
      i_allocated[i],  /* is allocated ? */
      i_count[i],      /* don't know what this is */
      i_lock[i],       /* not used */
      i_state[i],      /* not used */
      i_mutex[i];      /* used in create(): down(parent.d_inode->i_mutex)) */

    partition(3*nd-2+i:
      i_allocated[i]:
      i_count[i]:
      i_lock[i]:
      i_state[i]:
      i_mutex[i]
    );

    init(i_mutex[i]:1);            /* initially mutex available */
    init(i_lock[i]:1);  /* initially lock available */
  }

  //==========================================
  /* Processes */
  //==========================================

  /* Cleanup process */

  place
    p_start_cleanup_d,
    p_end_cleanup_d,
    p_start_cleanup_i,
    p_end_cleanup_i;
  partition(
    3*nd-2+ni+8*np+2:p_start_cleanup_d:p_end_cleanup_d:
      p_start_cleanup_i:p_end_cleanup_i
  );
  init(p_start_cleanup_d:1);
  trans
    t_start_cleanup_d,
    t_end_cleanup_d,
    t_start_cleanup_i,
    t_end_cleanup_i;

  for (int i in {1..nd}) {
    place p_cleanup_d[i];
    partition(3*nd-2+ni+8*np+2:p_cleanup_d[i]);
    trans
      t_skip_d[i],
      t_cleanup_d[i];
  }
  for (int i in {1..ni}) {
    place p_cleanup_i[i];
    partition(3*nd-2+ni+8*np+2:p_cleanup_i[i]);
    trans
      t_skip_i[i],
      t_cleanup_i[i];
  }
  arcs(
    p_start_cleanup_d:t_start_cleanup_d,
      t_start_cleanup_d:p_cleanup_d[1],
      dcache_lock:t_start_cleanup_d,
    p_cleanup_d[nd]:t_cleanup_d[nd],
      t_cleanup_d[nd]:p_end_cleanup_d,
      d_allocated[nd]:t_cleanup_d[nd]:tk(d_allocated[nd]),
      d_parent[nd]:t_cleanup_d[nd]:tk(d_parent[nd]),
      d_inode[nd]:t_cleanup_d[nd]:tk(d_inode[nd]),
      d_lock[nd]:t_cleanup_d[nd]:tk(d_lock[nd]),
      d_subdirs[nd]:t_cleanup_d[nd]:tk(d_subdirs[nd]),
      t_cleanup_d[nd]:d_lock[nd],
    p_cleanup_d[nd]:t_skip_d[nd],
      t_skip_d[nd]:p_end_cleanup_d,
    p_end_cleanup_d:t_end_cleanup_d,
      t_end_cleanup_d:p_start_cleanup_i,
      t_end_cleanup_d:dcache_lock
  );
  inhibit(dcache_lock:t_end_cleanup_d);
  guard(
    t_cleanup_d[nd]:tk(d_count[nd])==0,
    t_skip_d[nd]:tk(d_count[nd])>0
  );

  arcs(
    p_start_cleanup_i:t_start_cleanup_i,
      t_start_cleanup_i:p_cleanup_i[1],
      inode_lock:t_start_cleanup_i,
    p_cleanup_i[ni]:t_cleanup_i[ni],
      t_cleanup_i[ni]:p_end_cleanup_i,
    p_cleanup_i[ni]:t_skip_i[ni],
      t_skip_i[ni]:p_end_cleanup_i,
      i_allocated[ni]:t_cleanup_i[ni]:tk(i_allocated[ni]),
      i_lock[ni]:t_cleanup_i[ni]:tk(i_lock[ni]),
      i_state[ni]:t_cleanup_i[ni]:tk(i_state[ni]),
      i_mutex[ni]:t_cleanup_i[ni]:tk(i_mutex[ni]),
      t_cleanup_i[ni]:i_lock[ni],
      t_cleanup_i[ni]:i_mutex[ni],
    p_end_cleanup_i:t_end_cleanup_i,
      t_end_cleanup_i:p_start_cleanup_d,
      t_end_cleanup_i:inode_lock
  );
  inhibit(inode_lock:t_end_cleanup_i);
  guard(
    t_cleanup_i[ni]:tk(i_count[ni])==0,
    t_skip_i[ni]:tk(i_count[ni])>0
  );

  for (int i in {1..nd-1}) {
    arcs(
      p_cleanup_d[i]:t_cleanup_d[i],
        t_cleanup_d[i]:p_cleanup_d[i+1],
        d_allocated[i]:t_cleanup_d[i]:tk(d_allocated[i]),
        d_parent[i]:t_cleanup_d[i]:tk(d_parent[i]),
        d_inode[i]:t_cleanup_d[i]:tk(d_inode[i]),
        d_lock[i]:t_cleanup_d[i]:tk(d_lock[i]),
        d_subdirs[i]:t_cleanup_d[i]:tk(d_subdirs[i]),
        t_cleanup_d[i]:d_lock[i],
      p_cleanup_d[i]:t_skip_d[i],
        t_skip_d[i]:p_cleanup_d[i+1]
    );
    guard(
      t_cleanup_d[i]:tk(d_count[i])==0,
      t_skip_d[i]:tk(d_count[i])>0
    );
  }

  for (int i in {1..ni-1}) {
    arcs(
      p_cleanup_i[i]:t_cleanup_i[i],
        t_cleanup_i[i]:p_cleanup_i[i+1],
        i_allocated[i]:t_cleanup_i[i]:tk(i_allocated[i]),
        i_lock[i]:t_cleanup_i[i]:tk(i_lock[i]),
        i_state[i]:t_cleanup_i[i]:tk(i_state[i]),
        i_mutex[i]:t_cleanup_i[i]:tk(i_mutex[i]),
        t_cleanup_i[i]:i_lock[i],
        t_cleanup_i[i]:i_mutex[i],
      p_cleanup_i[i]:t_skip_i[i],
        t_skip_i[i]:p_cleanup_i[i+1]
    );
    guard(
      t_cleanup_i[i]:tk(i_count[i])==0,
      t_skip_i[i]:tk(i_count[i])>0
    );
  }



  //----------------------------------------
  //      Concurrent processes
  //----------------------------------------

  for (int p in {1..np}) {

    place
      /* "program counters" */
      p_begin[p],
      p_start_create[p],
      p_start_unlink[p],
      p_start_mkdir[p],
      p_start_rmdir[p],
      p_file[p],
      p_parent[p],
      p_inode[p];

    init(p_begin[p]:1);

    partition(
      3*nd-2+ni+8*p:p_begin[p]:p_start_create[p]:p_start_unlink[p]:p_start_mkdir[p]:p_start_rmdir[p],
      3*nd-2+ni+8*p-1:p_parent[p],
      3*nd-2+ni+8*p-2:p_file[p],
      3*nd-2+ni+8*p-3:p_inode[p]
    );

    trans
      t_start_create[p],
      t_start_unlink[p],
      t_start_mkdir[p],
      t_start_rmdir[p];
    arcs(
      p_begin[p]:t_start_create[p],
        t_start_create[p]:p_start_create[p],
        /* clear old arguments */
        p_file[p]:t_start_create[p]:tk(p_file[p]),
        p_parent[p]:t_start_create[p]:tk(p_parent[p]),
        p_inode[p]:t_start_create[p]:tk(p_inode[p]),
      p_begin[p]:t_start_unlink[p],
        t_start_unlink[p]:p_start_unlink[p],
        /* clear old arguments */
        p_file[p]:t_start_unlink[p]:tk(p_file[p]),
        p_parent[p]:t_start_unlink[p]:tk(p_parent[p]),
        p_inode[p]:t_start_unlink[p]:tk(p_inode[p]),
      p_begin[p]:t_start_mkdir[p],
        t_start_mkdir[p]:p_start_mkdir[p],
        /* clear old arguments */
        p_file[p]:t_start_mkdir[p]:tk(p_file[p]),
        p_parent[p]:t_start_mkdir[p]:tk(p_parent[p]),
        p_inode[p]:t_start_mkdir[p]:tk(p_inode[p]),
      p_begin[p]:t_start_rmdir[p],
        t_start_rmdir[p]:p_start_rmdir[p],
        /* clear old arguments */
        p_file[p]:t_start_rmdir[p]:tk(p_file[p]),
        p_parent[p]:t_start_rmdir[p]:tk(p_parent[p]),
        p_inode[p]:t_start_rmdir[p]:tk(p_inode[p])
    );


    /* ---------------------------------- */
    /* PN "program counters"  for create() */
    /* ---------------------------------- */
    place
      p_create_lookup[p],       // path_lookup(parent,file)
      p_create_line1[p],        // if (!parent.is_allocated)
      p_create_line2[p],        //  return ERROR
      p_create_line3[p],        // down(parent.d_inode->i_mutex)
      p_create_line4[p],        // if (file.is_allocated && !is_directory(file))
      p_create_line5[p],        //  up(parent.d_inode->i_mutex)
      p_create_line6[p],        //  path_release(file)
      p_create_line7[p],        //  return SUCCESS
      p_create_line8[p],        // if (file.is_alocated && is_directory(file))
      p_create_line9[p],        //  up(parent.d_inode->i_mutex)
      p_create_line10[p],       //  path_release(file)
      p_create_line11[p],       //  return ERROR
      p_create_line12[p],       // spin_lock(dcache_lock)
      p_create_line13[p],       // file = allocate_dentry()
      p_create_line14[p],       // if (file.is_allocated)
      p_create_line15[p],       //  spin_unlock(dcache_loc)
      p_create_line16[p],       //  up(parent.d_inode->i_mutex)
      p_create_line17[p],       //  dput(parent)
      p_create_line18[p],       //  return ERROR
      p_create_line19[p],       // dget(file)
      p_create_line20[p],       // spin_lock(inode_lock)
      p_create_line21[p],       // itmp = allocate_inode(file)
      p_create_line22[p],       // file.d_inode = &itmp
      p_create_line23[p],       // spin_unlock(inode_lock)
      p_create_line24[p],       // if (file.d_inode->is_allocated)
      p_create_line25[p],       //  atomic_write(d_count)
      p_create_line26[p],       //  dput(parent)
      p_create_line27[p],       //  spin_unlock(dcache_lock)
      p_create_line28[p],       //  up(parent.d_inode->i_mutex)
      p_create_line29[p],       //  return ERROR
      p_create_line30[p],       // update(parent)
      p_create_line31[p],       // path_release(file)
      p_create_line32[p],       // spin_unlock(dcache_lock)
      p_create_line33[p],       // up(parent.d_inode->i_mutex)
      p_create_line34[p];       // return SUCCESS
    partition(
      3*nd-2+ni+8*p-4:
       p_create_line1[p]:p_create_line2[p]:p_create_line3[p]:
       p_create_line4[p]:p_create_line5[p]:p_create_line6[p]:
       p_create_line7[p]:p_create_line8[p]:p_create_line9[p]:
       p_create_line10[p]:p_create_line11[p]:p_create_line12[p]:
       p_create_line13[p]:p_create_line14[p]:p_create_line15[p]:
       p_create_line16[p]:p_create_line17[p]:p_create_line18[p]:
       p_create_line19[p]:p_create_line20[p]:p_create_line21[p]:
       p_create_line22[p]:p_create_line23[p]:p_create_line24[p]:
       p_create_line25[p]:p_create_line26[p]:p_create_line27[p]:
       p_create_line28[p]:p_create_line29[p]:p_create_line30[p]:
       p_create_line31[p]:p_create_line32[p]:p_create_line33[p]:
       p_create_line34[p]:p_create_lookup[p]
    );

    /* ----- Create transitions ----- */

    /* ---------- Create step 0 ---------- */
    // --- initiate call: store new params
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step0[p][i][j];
        arcs(
          p_start_create[p]:t_create_step0[p][i][j],
            t_create_step0[p][i][j]:p_create_lookup[p],
            /* store new values */
            p_file[p]:t_create_step0[p][i][j]:tk(p_file[p]),
            p_parent[p]:t_create_step0[p][i][j]:tk(p_parent[p]),
            t_create_step0[p][i][j]:p_file[p]:i,
            t_create_step0[p][i][j]:p_parent[p]:j
        );
        inhibit(p_create_lookup[p]:t_create_step0[p][i][j]);
      }
    }
    /* ---------- Create: lookup ---------- */
    // --- path_lookup(parent,file)
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_lookup[p][i][j];
        arcs(
          p_create_lookup[p]:t_create_lookup[p][i][j],
            t_create_lookup[p][i][j]:p_create_line1[p],
            t_create_lookup[p][i][j]:d_count[i]:cond(tk(d_allocated[i])>0, 1, 0)
        );
        cond(i!=j, arcs(
            t_create_lookup[p][i][j]:d_count[j]:cond(tk(d_allocated[j])>0, 1, 0)), null);
        guard(
          t_create_lookup[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j
        );
        inhibit(d_count[i]:t_create_lookup[p][i][j]:nd);
        cond(i!=j, inhibit(d_count[j]:t_create_lookup[p][i][j]:nd), null);
      }
    }
    /* ---------- Create step 1 ---------- */
    // --- if (!parent.is_allocated)
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step1_then[p][i][j],
          t_create_step1_else[p][i][j];
        arcs(
          p_create_line1[p]:t_create_step1_then[p][i][j],
            t_create_step1_then[p][i][j]:p_create_line2[p],
            d_count[i]:t_create_step1_then[p][i][j]:cond(tk(d_allocated[i])>0, 1, 0),
          p_create_line1[p]:t_create_step1_else[p][i][j],
            t_create_step1_else[p][i][j]:p_create_line3[p]
        );
        guard(
          t_create_step1_then[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j & tk(d_allocated[j])==0,
          t_create_step1_else[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j & tk(d_allocated[j])>0
        );
      }
    }
    /* ---------- Create step 2 ---------- */
    // --- return ERROR
        trans
          t_create_step2[p];
        arcs(
          p_create_line2[p]:t_create_step2[p],
            t_create_step2[p]:p_begin[p]
        );
        inhibit(p_begin[p]:t_create_step2[p]);
    /* ---------- Create step 3 ---------- */
    // --- down(parent.d_inode->i_mutex)
    for (int j in {1..nd}) {
      for (int k in {1..ni}) {
        trans
          t_create_step3[p][j][k];
        arcs(
          p_create_line3[p]:t_create_step3[p][j][k],
            t_create_step3[p][j][k]:p_create_line4[p],
            i_mutex[k]:t_create_step3[p][j][k]
        );
        guard(
          t_create_step3[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
        );
      }
    }
    /* ---------- Create step 4 ---------- */
    // --- if (file.is_allocated && !is_directory(file))
    for (int i in {1..nd}) {
        trans
          t_create_step4_then[p][i],
          t_create_step4_else[p][i];
        arcs(
          p_create_line4[p]:t_create_step4_then[p][i],
            t_create_step4_then[p][i]:p_create_line5[p],
          p_create_line4[p]:t_create_step4_else[p][i],
            t_create_step4_else[p][i]:p_create_line8[p]
        );
        guard(
          t_create_step4_then[p][i]:tk(p_file[p])==i &   tk(d_allocated[i])>0 & tk(d_subdirs[i])==0,
          t_create_step4_else[p][i]:tk(p_file[p])==i & !(tk(d_allocated[i])>0 & tk(d_subdirs[i])==0)
        );
    }
    /* ---------- Create step 5 ---------- */
    // --- up(parent.d_inode->i_mutex)
    for (int j in {1..nd}) {
      for (int k in {1..ni}) {
        trans
          t_create_step5[p][j][k];
        arcs(
          p_create_line5[p]:t_create_step5[p][j][k],
            t_create_step5[p][j][k]:p_create_line6[p],
            t_create_step5[p][j][k]:i_mutex[k]
        );
        guard(
          t_create_step5[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
        );
        inhibit(i_mutex[k]:t_create_step5[p][j][k]:np);
      }
    }
    /* ---------- Create step 6 ---------- */
    // --- path_release(file)
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step6[p][i][j];
        arcs(
          p_create_line6[p]:t_create_step6[p][i][j],
            t_create_step6[p][i][j]:p_create_line7[p],
            d_count[i]:t_create_step6[p][i][j]:cond(tk(d_count[i])>1, 1, 0)
        );
        cond(i!=j, arcs(
            d_count[j]:t_create_step6[p][i][j]:cond(tk(d_count[j])>1, 1, 0)), null);
        guard(
          t_create_step6[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j
        );
      }
    }
    /* ---------- Create step 7 ---------- */
    // --- return SUCCESS
        trans
          t_create_step7[p];
        arcs(
          p_create_line7[p]:t_create_step7[p],
            t_create_step7[p]:p_begin[p]
        );
        inhibit(p_begin[p]:t_create_step7[p]);
    /* ---------- Create step 8 ---------- */
    // --- if (file.is_allocated && is_directory(file))
    for (int i in {1..nd}) {
        trans
          t_create_step8_then[p][i],
          t_create_step8_else[p][i];
        arcs(
          p_create_line8[p]:t_create_step8_then[p][i],
            t_create_step8_then[p][i]:p_create_line9[p],
          p_create_line8[p]:t_create_step8_else[p][i],
            t_create_step8_else[p][i]:p_create_line12[p]
        );
        guard(
          t_create_step8_then[p][i]:tk(p_file[p])==i &   tk(d_allocated[i])>0 & tk(d_subdirs[i])>0,
          t_create_step8_else[p][i]:tk(p_file[p])==i & !(tk(d_allocated[i])>0 & tk(d_subdirs[i])>0)
        );
    }
    /* ---------- Create step 9 ---------- */
    // --- up(parent.d_inode->i_mutex)
    for (int j in {1..nd}) {
      for (int k in {1..ni}) {
        trans
          t_create_step9[p][j][k];
        arcs(
          p_create_line9[p]:t_create_step9[p][j][k],
            t_create_step9[p][j][k]:p_create_line10[p],
            t_create_step9[p][j][k]:i_mutex[k]
        );
        guard(
          t_create_step9[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
        );
        inhibit(i_mutex[k]:t_create_step9[p][j][k]:np);
      }
    }
    /* ---------- Create step 10 ---------- */
    // --- path_release(file)
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step10[p][i][j];
        arcs(
          p_create_line10[p]:t_create_step10[p][i][j],
            t_create_step10[p][i][j]:p_create_line11[p],
            d_count[i]:t_create_step10[p][i][j]:cond(tk(d_count[i])>1, 1, 0)
        );
        cond(i!=j, arcs(
            d_count[j]:t_create_step10[p][i][j]:cond(tk(d_count[j])>1, 1, 0)), null);
        guard(
          t_create_step10[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j
        );
      }
    }
    /* ---------- Create step 11 ---------- */
    // --- return SUCCESS
        trans
          t_create_step11[p];
        arcs(
          p_create_line11[p]:t_create_step11[p],
            t_create_step11[p]:p_begin[p]
        );
        inhibit(p_begin[p]:t_create_step11[p]);
    /* ---------- Create step 12 ---------- */
    // --- spin_lock(dcache_lock)
        trans
          t_create_step12[p];
        arcs(
          p_create_line12[p]:t_create_step12[p],
            t_create_step12[p]:p_create_line13[p],
            dcache_lock:t_create_step12[p]
        );
    /* ---------- Create step 13 ---------- */
    // --- allocate_dentry
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step13[p][i][j];
        arcs(
          p_create_line13[p]:t_create_step13[p][i][j],
            t_create_step13[p][i][j]:p_create_line14[p],
            d_allocated[i]:t_create_step13[p][i][j]:tk(d_allocated[i]),
            t_create_step13[p][i][j]:d_allocated[i],
            d_count[i]:t_create_step13[p][i][j]:tk(d_count[i]),
            t_create_step13[p][i][j]:d_count[i],
            d_lock[i]:t_create_step13[p][i][j]:tk(d_lock[i]),
            t_create_step13[p][i][j]:d_lock[i],
            d_parent[i]:t_create_step13[p][i][j]:tk(d_parent[i]),
            t_create_step13[p][i][j]:d_parent[i]:j
        );
        guard(
          t_create_step13[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j & tk(d_allocated[i])==0
        );
      }
    }
    /* ---------- Create step 14 ---------- */
    // --- if (!file.is_allocated)
    for (int i in {1..nd}) {
        trans
          t_create_step14_then[p][i],
          t_create_step14_else[p][i];
        arcs(
          p_create_line14[p]:t_create_step14_then[p][i],
            t_create_step14_then[p][i]:p_create_line15[p],
          p_create_line14[p]:t_create_step14_else[p][i],
            t_create_step14_else[p][i]:p_create_line19[p]
        );
        guard(
          t_create_step14_then[p][i]:tk(p_file[p])==i & tk(d_allocated[i])==0,
          t_create_step14_else[p][i]:tk(p_file[p])==i & tk(d_allocated[i])>0
        );
    }
    /* ---------- Create step 15 ---------- */
    // --- spin_unlock(dcache_lock)
        trans
          t_create_step15[p];
        arcs(
          p_create_line15[p]:t_create_step15[p],
            t_create_step15[p]:p_create_line16[p],
            t_create_step15[p]:dcache_lock
        );
        inhibit(dcache_lock:t_create_step15[p]:2);
    /* ---------- Create step 16 ---------- */
    // --- up(parent.d_inode->i_mutex)
    for (int j in {1..nd}) {
      for (int k in {1..ni}) {
        trans
          t_create_step16[p][j][k];
        arcs(
          p_create_line16[p]:t_create_step16[p][j][k],
            t_create_step16[p][j][k]:p_create_line17[p],
            t_create_step16[p][j][k]:i_mutex[k]
        );
        guard(
          t_create_step16[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
        );
        inhibit(i_mutex[k]:t_create_step16[p][j][k]:np);
      }
    }
    /* ---------- Create step 17 ---------- */
    // --- dput(parent)
    for (int j in {1..nd}) {
        trans
          t_create_step17[p][j];
        arcs(
          p_create_line17[p]:t_create_step17[p][j],
            t_create_step17[p][j]:p_create_line18[p],
            d_count[j]:t_create_step17[p][j]:cond(tk(d_count[j])>1, 1, 0)
        );
        guard(
          t_create_step17[p][j]:tk(p_parent[p])==j
        );
    }
    /* ---------- Create step 18 ---------- */
    // --- return ERROR
        trans
          t_create_step18[p];
        arcs(
          p_create_line18[p]:t_create_step18[p],
            t_create_step18[p]:p_begin[p]
        );
        inhibit(p_begin[p]:t_create_step18[p]);
    /* ---------- Create step 19 ---------- */
    // --- dget(file)
    for (int i in {1..nd}) {
        trans
          t_create_step19[p][i];
        arcs(
          p_create_line19[p]:t_create_step19[p][i],
            t_create_step19[p][i]:p_create_line20[p],
            t_create_step19[p][i]:d_count[i]
        );
        guard(
          t_create_step19[p][i]:tk(p_file[p])==i
        );
        inhibit(d_count[i]:t_create_step19[p][i]:np+1);
    }
    /* ---------- Create step 20 ---------- */
    // --- spin_lock(inode_lock)
        trans
          t_create_step20[p];
        arcs(
          p_create_line20[p]:t_create_step20[p],
            t_create_step20[p]:p_create_line21[p],
            inode_lock:t_create_step20[p]
        );
    /* ---------- Create step 21 ---------- */
    // --- allocate_inode(file)
    for (int k in {2..ni}) {
        trans
          t_create_step21[p][k];
        arcs(
          p_create_line21[p]:t_create_step21[p][k],
            t_create_step21[p][k]:p_create_line22[p],
            t_create_step21[p][k]:i_allocated[k],
            i_count[k]:t_create_step21[p][k]:tk(i_count[k]),
            t_create_step21[p][k]:i_count[k],
            i_mutex[k]:t_create_step21[p][k]:tk(i_mutex[k]),
            t_create_step21[p][k]:i_mutex[k],
            i_lock[k]:t_create_step21[p][k]:tk(i_lock[k]),
            t_create_step21[p][k]:i_lock[k],
            p_inode[p]:t_create_step21[p][k]:tk(p_inode[p]),
            t_create_step21[p][k]:p_inode[p]:k
        );
        guard(
          t_create_step21[p][k]:tk(i_allocated[k])==0
        );
        inhibit(i_allocated[k]:t_create_step21[p][k]);
        /* allocate first inode available */
        /* i.e. if allocating inode k, then all inodes before k are already allocated */
        for (int h in {1..k-1}) {
          arcs(
            i_allocated[h]:t_create_step21[p][k],
            t_create_step21[p][k]:i_allocated[h]
          );
        }
    }
        /* no inodes available */
        trans
          t_create_step21x[p];
        arcs(
          p_create_line21[p]:t_create_step21x[p],
            t_create_step21x[p]:p_create_line22[p],
            p_inode[p]:t_create_step21x[p]:tk(p_inode[p]),
            t_create_step21x[p]:p_inode[p]:ni+1
        );
        for (int h in {1..ni}) {
          arcs(
            i_allocated[h]:t_create_step21x[p],
            t_create_step21x[p]:i_allocated[h]
          );
        }
    /* ---------- Create step 22 ---------- */
    // --- file.d_inode = &itmp
    for (int i in {1..nd}) {
      for (int k in {2..ni+1}) {
        trans
          t_create_step22[p][i][k];
        arcs(
          p_create_line22[p]:t_create_step22[p][i][k],
            t_create_step22[p][i][k]:p_create_line23[p],
            d_inode[i]:t_create_step22[p][i][k]:tk(d_inode[i]),
            t_create_step22[p][i][k]:d_inode[i]:k
        );
        guard(
          t_create_step22[p][i][k]:tk(p_file[p])==i & tk(p_inode[p])==k
        );
      }
    }
    /* ---------- Create step 23 ---------- */
    // --- spin_unlock(inode_lock)
        trans
          t_create_step23[p];
        arcs(
          p_create_line23[p]:t_create_step23[p],
            t_create_step23[p]:p_create_line24[p],
            t_create_step23[p]:inode_lock
        );
        inhibit(inode_lock:t_create_step23[p]);
    /* ---------- Create step 24 ---------- */
    // --- if (!file.d_inode->is_allocated)
    for (int k in {1..ni}) {
        trans
          t_create_step24_then[p][k],
          t_create_step24_else[p][k];
        arcs(
          p_create_line24[p]:t_create_step24_then[p][k],
            t_create_step24_then[p][k]:p_create_line25[p],
          p_create_line24[p]:t_create_step24_else[p][k],
            t_create_step24_else[p][k]:p_create_line30[p]
        );
        guard(
          t_create_step24_then[p][k]:tk(p_inode[p])==k & tk(i_allocated[k])==0,
          t_create_step24_else[p][k]:tk(p_inode[p])==k & tk(i_allocated[k])>0
        );
    }
        /* none available */
        trans
          t_create_step24_thenx[p];
        arcs(
          p_create_line24[p]:t_create_step24_thenx[p],
            t_create_step24_thenx[p]:p_create_line25[p]
        );
        guard(
          t_create_step24_thenx[p]:tk(p_inode[p])==ni+1
        );
    /* ---------- Create step 25 ---------- */
    // --- atomic_write(file.d_count, 0)
    for (int i in {1..nd}) {
        trans
          t_create_step25[p][i];
        arcs(
          p_create_line25[p]:t_create_step25[p][i],
            t_create_step25[p][i]:p_create_line26[p],
            d_count[i]:t_create_step25[p][i]:tk(d_count[i]),
            d_allocated[i]:t_create_step25[p][i]:tk(d_allocated[i]),
            d_parent[i]:t_create_step25[p][i]:tk(d_parent[i])
        );
        guard(
          t_create_step25[p][i]:tk(p_file[p])==i
        );
    }
    /* ---------- Create step 26 ---------- */
    // --- dput(parent)
    for (int j in {1..nd}) {
        trans
          t_create_step26[p][j];
        arcs(
          p_create_line26[p]:t_create_step26[p][j],
            t_create_step26[p][j]:p_create_line27[p],
            d_count[j]:t_create_step26[p][j]:cond(tk(d_count[j])>1, 1, 0)
        );
        guard(
          t_create_step26[p][j]:tk(p_parent[p])==j
        );
    }
    /* ---------- Create step 27 ---------- */
    // --- spin_unlock(dcache_lock)
        trans
          t_create_step27[p];
        arcs(
          p_create_line27[p]:t_create_step27[p],
            t_create_step27[p]:p_create_line28[p],
            t_create_step27[p]:dcache_lock
        );
        inhibit(dcache_lock:t_create_step27[p]);
    /* ---------- Create step 28 ---------- */
    // --- up(parent.d_inode->i_mutex)
    for (int j in {1..nd}) {
      for (int k in {1..ni}) {
        trans
          t_create_step28[p][j][k];
        arcs(
          p_create_line28[p]:t_create_step28[p][j][k],
            t_create_step28[p][j][k]:p_create_line29[p],
            t_create_step28[p][j][k]:i_mutex[k]
        );
        guard(
          t_create_step28[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
        );
        inhibit(i_mutex[k]:t_create_step28[p][j][k]);
      }
    }
    /* ---------- Create step 29 ---------- */
    // --- return ERROR
        trans
          t_create_step29[p];
        arcs(
          p_create_line29[p]:t_create_step29[p],
            t_create_step29[p]:p_begin[p]
        );
        inhibit(p_begin[p]:t_create_step29[p]);
    /* ---------- Create step 30 ---------- */
    // --- update_parent()
    for (int i in {1..nd}) {
      for (int j in {1..nd}) {
        trans
          t_create_step30[p][i][j];
        arcs(
          p_create_line30[p]:t_create_step30[p][i][j],
            t_create_step30[p][i][j]:p_create_line31[p],
            d_parent[i]:t_create_step30[p][i][j]:tk(d_parent[i]),
            t_create_step30[p][i][j]:d_parent[i]:j,
            t_create_step30[p][i][j]:d_subdirs[j]
        );
        guard(
          t_create_step30[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j
        );
        inhibit(d_subdirs[j]:t_create_step30[p][i][j]:nd);
      }
    }
    /* ---------- Create step 31 ---------- */
    // --- path_release(file)
    for (int i in {1..nd}) {
        for (int j in {1..nd}) {
          trans
            t_create_step31[p][i][j];
          arcs(
            p_create_line31[p]:t_create_step31[p][i][j],
              t_create_step31[p][i][j]:p_create_line32[p],
              d_count[i]:t_create_step31[p][i][j]:cond(tk(d_count[i])>1, 1, 0)
          );
          cond(i!=j, arcs(
              d_count[j]:t_create_step31[p][i][j]:cond(tk(d_count[j])>1, 1, 0)), null);
          guard(
            t_create_step31[p][i][j]:tk(p_file[p])==i & tk(p_parent[p])==j
          );
        }
      }
      /* ---------- Create step 32 ---------- */
      // --- spin_unlock(dcache_lock)
          trans
            t_create_step32[p];
          arcs(
            p_create_line32[p]:t_create_step32[p],
              t_create_step32[p]:p_create_line33[p],
              t_create_step32[p]:dcache_lock
          );
          inhibit(dcache_lock:t_create_step32[p]);
      /* ---------- Create step 33 ---------- */
      // --- up(parent.d_inode->i_mutex)
      for (int j in {1..nd}) {
        for (int k in {1..ni}) {
          trans
            t_create_step33[p][j][k];
          arcs(
            p_create_line33[p]:t_create_step33[p][j][k],
              t_create_step33[p][j][k]:p_create_line34[p],
              t_create_step33[p][j][k]:i_mutex[k]
          );
          guard(
            t_create_step33[p][j][k]:tk(p_parent[p])==j & tk(d_inode[j])==k
          );
          inhibit(i_mutex[k]:t_create_step33[p][j][k]);
        }
      }
      /* ---------- Create step 34 ---------- */
      // --- return SUCCESS
          trans
            t_create_step34[p];
          arcs(
            p_create_line34[p]:t_create_step34[p],
              t_create_step34[p]:p_begin[p]
          );
          inhibit(p_begin[p]:t_create_step34[p]);


    [...]
    [Unlink code]
    [Mkdir code]
    [Rmdir code]
    [...]

    } // ============ end process p


    // initialization
    init(
      /* root node is created at mount
         has itself and lost+found as subdirs
         has itself as parent */
        d_allocated[ROOT]:1,
        d_inode[ROOT]:ROOT,
        d_parent[ROOT]:ROOT,
        d_subdirs[ROOT]:2,
        d_count[ROOT]:1,
        i_count[ROOT]:1,
        i_allocated[ROOT]:1,
        /* lost+found node is created at mount
           has itself as subdir
           has root as parent */
        d_allocated[LOST_FOUND]:1,
        d_inode[LOST_FOUND]:LOST_FOUND,
        d_parent[LOST_FOUND]:ROOT,
        d_subdirs[LOST_FOUND]:1,
        d_count[LOST_FOUND]:1,
        i_count[LOST_FOUND]:1,
        i_allocated[LOST_FOUND]:1
      );

      real ro   := reorder;
      bigint ns := num_states(false);
      bool db   := debug;

      stateset Deadlock := difference(reachable, prev(potential(true)));
      bigint  nDeadlock := card(Deadlock);
      bool    pDeadlock := printset(Deadlock);
      bool    tDeadlock := EFtrace(initialstate, Deadlock);

    };


    int ND := read_int("Number of d_entries");
    int NI := read_int("Number of i_nodes");
    int NP := read_int("Number of processes");
    compute(ND);
    compute(NI);
    compute(NP);

    print("******************************************************\n");
    print("*                 VFS abstract model                 *\n");
    print("******************************************************\n");
    print("* System parameters:\n");
    print("* - Dentry pool size:    ", ND, "\n");
    print("* - Inode pool size:     ", NI, "\n");
    print("* - Number of processes: ", NP, "\n");

    //compute(EXT2(ND,NI,NP).ro);
    compute(EXT2(ND,NI,NP).db);
    print("\nNumber of reachable states: ", EXT2(ND,NI,NP).ns, "\n");

    print("\nNumber of deadlocked states: ", EXT2(ND,NI,NP).nDeadlock);
    compute(EXT2(ND,NI,NP).pDeadlock);
    compute(EXT2(ND,NI,NP).tDeadlock);
```