# Simulating a Flash File System with CoreASM and Eclipse

Maximilian Junker
Fakultät für Informatik Technische Universität München
D-85748 Garching, Germany
Email: junkerm@in.tum.de

Dominik Haneberg, Gerhard Schellhorn, Wolfgang Reif, Gidon Ernst
Lehrstuhl für Softwaretechnik und Programmiersprachen Universität Augsburg
D-86135 Augsburg, Germany
Email: {haneberg,schellhorn,reif,ernst}@informatik.uni-augsburg.de

**Abstract:** The formal specification of a file system for flash memory is the first step towards its verification. But creating such a formal specification is complex and error-prone. Visualizing the system state and having an executable version of the specification helps to better understand the specified system. In this paper, we present an approach for simulating and visualizing specifications written in the Abstract State Machine (ASM) formalism. We extend the ASM execution engine CoreASM to execute ASMs written using algebraic specifications. Furthermore we develop an Eclipse-based visualization framework and integrate CoreASM into it. This enables us to create different abstract views of the CoreASM system state and allows the user to interact with the specification in an intuitive way. We apply our techniques to the visualization of an abstract specification of a flash memory file system and report on our experiences with CoreASM and Eclipse.

## 1 Introduction

The popularity of flash memory as storage device has been increasing constantly over the last years. Flash memory offers a couple of important advantages compared to magnetic storage: It has no moving parts and is therefore less susceptible to mechanical shock. Flash memory also offers a better energy efficiency. But there is a downside as well. Flash memory characteristics are considerably different from those of magnetic storage: Flash memory cannot be overwritten, but only erased in blocks and erasing should be done evenly ("wear leveling") because it wears out the flash cells after approx. $10^5$ erase cycles due to the high voltage that needs to be applied. These properties imply that standard file systems cannot be used with flash memory directly. One of the possible solutions for dealing with these special characteristics is to use a special flash file system (FFS for short) which is designed to with the specifics of flash memory in mind.

Since flash memory is beginning to be used in safety-critical applications, Joshi and Holzmann [JH07] from the NASA JPL proposed in 2007 the verification of a FFS as a project of Hoare's Verification Grand Challenge [Hoa03]. Their goal was a verified FFS for use in

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

future missions. NASA already uses flash memory in spacecrafts, e.g. on the Mars Exploration Rovers. This already had nearly disastrous consequences as the Mars Rover Spirit was almost lost due to an anomaly in the software access to the flash store [RN05]. For our contribution within the challenge we chose to base our work on the newest of the FFS of the Linux kernel: *UBIFS* [Hun08].

We currently develop formal models of the various layers of UBIFS. These will be connected via refinements, and we aim at a fully verified prototype implementation. A formal specification of the operations provided for the Linux Virtual File System (VFS) – that is used in Linux as an internal interface to all file systems – is given in [SSHR09], and several more layers (the Memory Technology Device (MTD) interface to low-level flash drivers, the UBI layer which maps logical to physical erase blocks etc.) are currently under development.

Our specification language is *Abstract State Machines* (ASM) [Gur95, BS03], which defines operations over abstract data types. We use algebraic specifications to axiomatise data types, and our interactive theorem KIV [RSSB98] to verify properties. ASMs prefer an operational style (using parallel assignments, nondeterministic choice etc.) that defines *rules* over purely relational notation as in Z schemata [Spi89]. This makes the task of getting executable operations for a simulation easier. ASM rules are similar to Event-B [Abr10], but they support all sequential programming constructs (e.g. sequential composition, while loops and recursive calls). Although ASMs use an easy to read "pseudocode" syntax, writing a correct specification is still error prone.

In this paper we report on an approach to simulate specifications using the *CoreASM* [FGG07] framework and Eclipse plugins for visualization. Using CoreASM to simulate abstract specifications gives several benefits:

- Using CoreASM allows us to prototype and test our specification. In particular it is possible to test if invariants are maintained while the CoreASM model is executed. Many problems with a specification may be found quickly before even starting expensive verification attempts.
- Simulation and visualization are good means to get an understanding of the inner workings of systems that use many complex data structures. Someone who is new to the specification can simply execute it in order to explore its different parts and how they interact.
- Formal specifications are usually hard to read. A visualization also works as a means of communication and may facilitate discussions about a specification.

The last two points are particularly relevant for file systems, where typical specifications (e.g. [MS87, DBA08, HL09]) provide the view of a directory tree. While these are adequate as top-level specifications, implementations of real file systems (e.g. ext2/3 for Linux or NTFS for Windows, also UBIFS and other flash file systems) are *not* organized around a directory tree as the central concept for actually storing files. They use more complex data structures like *inodes* and *journals*. The actual directory tree exists only as an abstraction.

Our simulation therefore has to show how these data structures of the implementation are


ISBN 978-3-88579-286-4

used. To do this it also computes the abstract directory tree by resolving links stored in nodes. Thereby it gives a prototype of the abstraction function we will need in order to verify that the data structures are indeed a refinement of the top-level view as a directory tree. Whether the implemented operations result in the expected effects on the abstract tree can already be observed in the visualization.

There are only few tools for simulating or executing abstract specifications. Outside the ASM world there is e.g. Jaza, a simulator for Z [Spi89] specifications, the AlloyAnalyzer for Alloy [Jac06] and ProB for B [Abr10] specifications. Jaza offers no GUI so a visualization as we envision it is not in its scope. The AlloyAnalyzer is for verifying properties of finite models, a user guided simulation is not its purpose. ProB offers similar possibilities for visualization as our CoreASM/Eclipse approach and would be a valid alternative. Using B would however require us to encode ASM syntax like sequential composition into sequences of events using a program counter. Using CoreASM allowed us to stay close to the syntax of the formal specification. There are also tools which offer ways of executing ASM specifications. Some compile to programming languages [Anl00, Gro03] and therefore do not offer interactive execution. ASM Gofer [Sch08] has only a limited GUI and no user input.

The rest of this paper is organized as follows. Section 2 gives a short overview of our specification of UBIFS from [SSHR09], its datatypes and operations. In Section 3 we show how we implement this specification in CoreASM and give details on our extensions to the CoreASM language. Section 4 describes our Eclipse-based visualization framework and how it is integrated with CoreASM. Finally, we summarize and draw conclusions in Section 5.

The complete simulation environment together with the file system specification is available for download from the project website[1].

## 2 Specification of a Flash File System

UBIFS is a FFS which is part of the Linux kernel since version 2.6.27 (released in 2008). It adheres to the flash memory characteristics like out-of-place updates and the need for garbage collection. Our specification in [SSHR09] gives a formal model of the UBIFS' interface to the Virtual File System (VFS) layer. This layer is between a POSIX-like view on the file system and the real flash hardware. The abstraction level of this specification is still high and many aspects of a FFS such as the mapping of data to erase blocks is not taken into account. The specification features the four core data structures of the implementation, the *flash store*, the *RAM index*, the *flash index* and the *journal* as well as a set of standard operations on a file system like to create, delete and rename files and directories, to make hard links as well as read and write operations.

Figure 1 shows these central data structures of UBIFS in the four columns. We give an overview how they are used by operations. The simulation we will give in the following

[1] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/Flashix/`

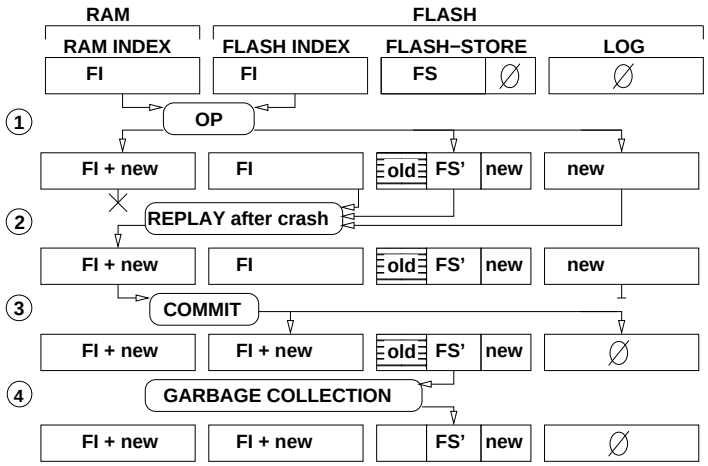



Figure 1: Impact of UBIFS operations on the data structures

sections shows the state of the file system (see Fig. 4) and allows interactive exploration of the precise effects of the operations. The flash store, shown in column 3, is used to store the raw data of a file system as a mapping of *addresses* to *nodes*. Nodes may store the content of files (pages), or directory entries. They never contain direct links (addresses), since this would make relocating data impossible. Instead they store symbolic *keys*.

The initial flash store in the topmost line contains some data $FS$ and some empty areas ($\emptyset$). In step (1) of the figure, a regular operation ($OP$) is performed, e.g. overwriting data in a file. The second line shows the new state: The flash store has some new data ($new$), some unchanged parts ($FS'$) and some obsolete data ($old$) due to the out-of-place updates. These obsolete data can be collected by garbage collection as shown in step (4).

A problem that is crucial for the efficiency of the FFS is indexing. Without an index that allows searching for a specific node efficiently, the whole media would have to be scanned. Therefore UBIFS uses an index that maps keys to the current address of the corresponding node on the flash media. The index in UBIFS is organized as a $B^+$-tree and stored in memory (*RAM index*). The index is stored on flash (*flash index*) as well, because rebuilding the index by scanning the complete media would take up a lot of time at mount time. The RAM and the flash index are shown as the two leftmost columns in Fig. 1. Having an index on flash poses some new difficulties as the index must be updated out-of-place on the flash media (this is done by using a wandering tree) and the index has to be changed every time old data is changed or new data is added (because the new data must be added to the index and the position of the modified data changes due to out-of-place updates). To limit the number of changes to the flash index, UBIFS does not update the flash index immediately, but uses a journal (also called *log*) of recent changes instead, which is shown in the rightmost column of Fig. 1. Its use is illustrated in the second line of Fig. 1: The log contains references to the new data written to the flash store in step (1) and the RAM index was updated accordingly ($FI + new$) while the flash index remained unchanged.

At certain points in time, the flash index is synchronized with the RAM index by the *commit* operation. This can be seen as step (3) in Fig. 1: The flash index is replaced with the current RAM index and the log is emptied.

Since fault tolerance is an important requirement for a FFS, one problem remains: What happens when the system crashes (e.g. power-failure) before the RAM index is written to flash? In this case, the flash index is out-of-date, compared to the data in the flash store. This problem is solved by the journal, which records all changes to the data on flash that have not yet been added to the flash index. A special operation, *replay*, is done after a system crash to recover an up-to-date RAM index (step (2) in Fig. 1): first, the flash index is read as preliminary RAM index. Then all changes that were recorded in the log are applied to rebuild the correct RAM index.

## 3 Translating the Specification to CoreASM

CoreASM [FGG07] is a language and a set of tools for the specification and execution of Abstract State Machines. The main aims of CoreASM are to maintain executability even of abstract models and to support extensions to the language in a flexible way. A specification written in the CoreASM language can be executed by the CoreASM *engine*. The engine maintains the current system state as an *abstract storage*, which is essentially a mapping between locations and values. The abstract storage may be changed by *updates* which are generated during execution. The engine interacts with the environment through a single interface which provides various operations such as: load a specification, start a run, perform a single step, get and set the value of a location, access the update set produced by a step, etc. The operations offered by the control API provide the foundation for integrating external tools with the execution engine, thus realizing an open tool architecture for ASMs. Due to the modular architecture of CoreASM most of the functionality of the engine is implemented through plugins to the basic kernel.

As our original specification was focussed on verification, certain adaptations were necessary to make the specification executable in CoreASM. We describe these in the following two paragraphs.

### 3.1 Translating Data Types

CoreASM comes with a set of predefined basic data structures like lists and maps. We use these to represent the store, the indices and the log. At the time of writing, some of CoreASM's built-in data structures were not complete yet. For example, we had to extended maps by operations for adding and removing elements.

Besides maps, our UBIFS specification heavily uses freely generated *algebraic data types* that are not directly supported by CoreASM. Figure 2 shows an exemplary definition of directory entries. Instances are created with one of the two constructors, accessed by selector functions and distinguished by predicates such as "dentry?".


ISBN 978-3-88579-286-4

```
dentry = mkdentry(name : string, ino : nat) with dentry?
       | negdentry(name : string) with negdentry?
```

Figure 2: Defintion of the dentry data type

We extended CoreASM with a plugin that adds the missing data types and function symbols. This plugin consists of two building blocks. First, it contains Java classes for each data type to represent its instances. These are derived from the CoreASM system class *Element*. Second, functions are implemented by a Java classes derived from the system class *FunctionElement*. They are registered at the engine under a given name (e.g. "mkdentry", "ino") during plugin initialization. The effect of calling such functions is given by overriding the method *Element getValue(List<Element> args)*.

The translation leads to a high number of function classes. Direct support for free data types in CoreASM would be preferable – which we have not addressed as it would require deep modifications, for example in the parser.

### 3.2 Translating Rules

With two notable exceptions, the rules of the ASM specification in KIV translate directly to rules of the CoreASM specification.

First, for simulation purpose, the user must be able to select which rules are executed. Originally, the *main* rule nondeterministically executes file system operations in a loop. The rule is changed to select operations requested by the user, as described in Section 4.

Second, the semantics of the *choose* construct in KIV is to nondeterministically select a value for a variable of a specific sort that satisfies some constraint. The range is always the (infinite) carrier set of the sort. To make choice executable, CoreASM requires to explicitly specify a "big" range from which the value is chosen as shown in Figure 3.

```
choose adr in [1..MAX_ADR] with not hasKey(FS, adr) do ...
```

Figure 3: Choosing a fresh address from a finite range

## 4 Visualizing the Specification

In this section we show how the CoreASM system state can be displayed within Eclipse and how visualization and execution engine communicate. We first give an architecture overview and then describe the different components.

Figure 4 shows the running simulation. The four tables in the middle correspond to the

data structures shown in Figure 1. The lower left window displays a tree view of the file system. User interaction is initiated through context menus.

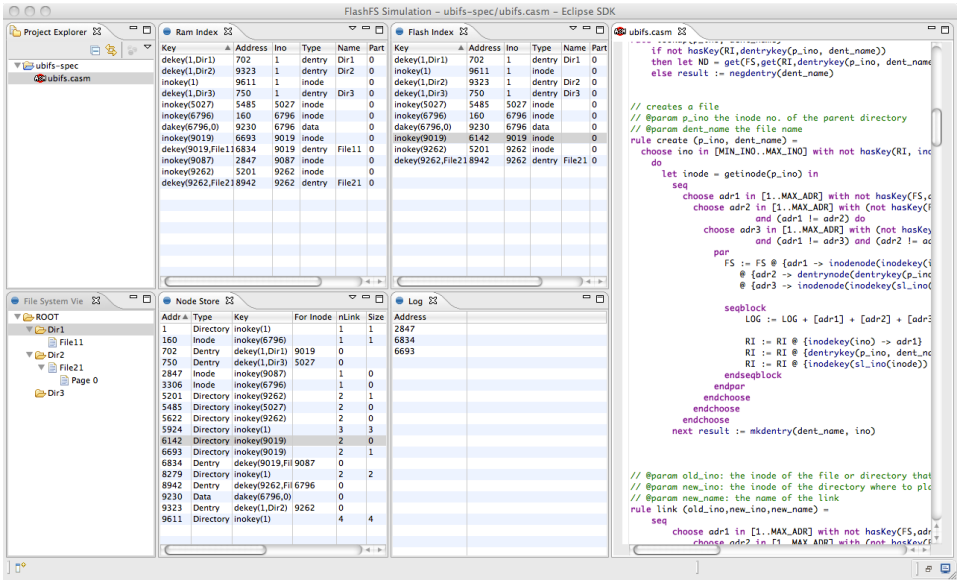



Figure 4: The simulation running our specification with some added directories and files.

### 4.1 Architecture

The main parts of our architecture are pictured in the UML component diagram in Figure 5. Each rectangular shape represents a component of our system. The components may communicate with each other over ports (small rectangular shapes) that provide interfaces to which other components may connect. Communication ports may be delegated to inner components which means that if a request is being made on a port it is not answered directly by the component but sent to the inner component which the port delegates to.

The heart of our application is the CoreASM engine. It executes the specification and maintains the abstract storage. As the CoreASM engine has a plugin structure we may contribute components to the CoreASM engine itself. In addition to the predefined *Abstract Storage*, there is the *FlashFS* plugin which contains the additions to CoreASM described in Section 3.1. Furthermore we implemented two components (*Storage Observer* and *Remote Command*) that enable the communication with the CoreASM engine from the outside. Each component is responsible for one direction of communication. These two components will be described in detail in Sections 4.2 and 4.3.

CoreASM comes with an Eclipse plugin that integrates the execution engine into Eclipse. We expose the two communication components via the standard Eclipse *extension point*

mechanism, so that they can be accessed easily by the visualization plugin which is responsible for the graphical display of the system state and the interaction.

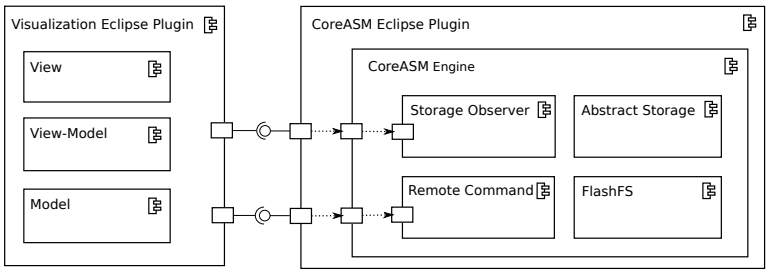



Figure 5: Architecture of the simulation environment as a UML component diagram

### 4.2 Visualization

As our specification is concerned with a file system, the most obvious visualization is a file system tree. It is an interpretation of the current state of the RAM index and the node store and contains directories, files and for each file its data pages. It can be navigated in the usual way. The tree view is the most abstract view on the simulated file system. The RAM index, flash index, node store and log are visualized in a tabular way where each row represents a single entry in the index or store. The tree is visually linked to the tables: if the user selects an element in the file system tree, the corresponding rows in the table views are highlighted. In this way the underlying data structure may be comfortably inspected. Typical file-system tasks (e.g. creating new files) as well as simulation related tasks (e.g. simulate a crash) may be executed via context menus. Each task corresponds directly to an ASM rule in the specification.

The visualization component contains a snapshot of the current CoreASM abstract storage (the *model* in Figure 5). In order to populate the visualization, the model is not suited as a direct source as it is not abstract enough. This gap between the model and the graphical view is bridged via *view-models*. The view-models are interpretations of the model and are connected to the graphical components (views) via *data binding*: as soon as the model changes, the view-models are updated automatically and so are the views.

The model itself must reflect the current state of the simulation (i.e. the abstract storage). The model is therefore connected to the CoreASM component *Storage Observer* via the publish-subscribe pattern. Whenever the abstract storage within the CoreASM engine is changed, this component notifies all subscribers via their callback method *storageChange*, passing the modified locations.

Most of the program logic of the visualization is concerned with the mappings between model and view-models. As an example, Figure 6 shows a part of the algorithm that computes the view-model for the file system tree. The tree is made up of tree nodes which represent directories, files or pages. Note that the tree nodes are not identical to the nodes


ISBN 978-3-88579-286-4

of the store. The method *buildTree* keeps a list of tree nodes that it still has to visit. Initially the list only contains the root node. In each iteration the algorithm takes a tree node from the list, retrieves the node's children from the model (*getChildrenFor*) and inserts those in the working list. For brevity reasons we only show the case for retrieving the children of a file node (which are page nodes). By looking up the file's inode in the RAM index we can determine the address of the file's inode in the store. Now we can retrieve the node, which contains the number of pages that this file is made up of. We can now iteratively use the file's inode number and a page number to look up a page's address and retrieve the page node from the store. Note that the method *buildTree* is the prototype of an abstraction function for the formal verification.

```
// Builds the file system tree from the model
public FsTreeNode buildTree() {
  List<FsTreeStructureNode> toVisit = new ArrayList<FsTreeStructureNode>();
  FsTreeStructureNode rootNode = getRootNode();
  toVisit.add(rootNode);
  while(!toVisit.isEmpty()){
    FsTreeStructureNode next = toVisit.get(0);
    next.addChildren(getChildrenFor(next));
    toVisit.addAll(next.getChildren());
    toVisit.remove(next);
  }
  return rootNode;
}
// Retrieves the children of a node
public List<FsTreeNode> getChildrenFor(FsTreeStructureNode node) {
  List<FsTreeNode> childNodes = new ArrayList<FsTreeNode>();
  if(node.isDir()){...}
  else if(node.isFile()){
    FsTreeFileNode file = (FsTreeFileNode)node;
    List<NodeElement> pages = getPageNodesForFile(file.getInode());
    for(NodeElement p: pages){
     childNodes.add(new FsTreePageNode(p.getData().getContent(),p.getKey().getPart(),file));
    }}
  return childNodes;
}
// Retrieve the page nodes for a file given as inode
public List<NodeElement> getPageNodesForFile(Integer inode) {
  List<NodeElement> entries = new ArrayList<NodeElement>();
  // fetch the address of the inode from the ram index
  Integer address = getRamIndex().getAddress(new InodeKeyElement(inode));
  // fetch the node from the store
  NodeElement node = getStore().getNode(address);
  // get all page nodes
  if(node != null){
    for(int i = 0;i<node.getSize();i++){
      // fetch page node address
      Integer pAddress = getRamIndex().getAddress(new DataKeyElement(inode,i));
      // fetch page node
      NodeElement pnode = getStore().getNode(pAddress);
      if(pnode!=null)entries.add(pnode); }
  }
  return entries;
}
```

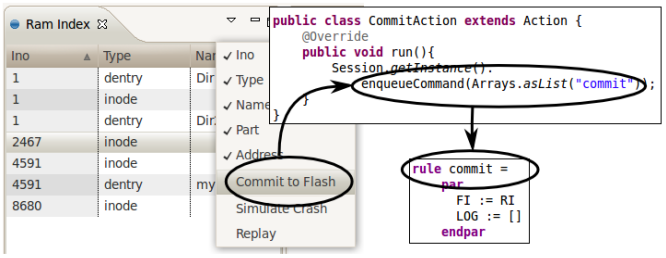
Figure 6: Part of the abstraction algorithm that maps the specification model to a file system tree. `FsTreeNode` and its variants are elements of the view-model. `NodeElement` is an entry in the store, which is part of the model.

### 4.3 Remote Command – Interaction with the Running Specification

The user has the possibility to interact with the specification via the UI. This essentially means that the next rule of the specification which is executed may be selected using the graphical interface.

All actions that the user may execute in the UI map to the execution of a rule of the specification. We propagate information about the selected rule down to the engine and the running specification using so-called *commands*. Such a command consists of a rule name and a list of arguments for the call.

The plugin *Remote Command* is responsible for this direction of the communication. It holds an internal queue of commands that should be executed in the next steps and provides a public method *enqueueCommand* to insert a command into this list. In the specification we may access the first command in the queue using the function $next\_command$. The main rule of the ASM then activates other rules depending on the command it found in the queue. On the visualization side, every action is implemented by simply enqueueing a command. The relation between the UI and ASM rules via commands is illustrated in Figure 7.




```
rule main =
  let cmd = next_command() in
      if head(cmd) = "commit" then
          commit
      else ...
```

Figure 7: Interaction between UI and specification. Every UI element that offers user interaction is in a 1:1 correspondence to a rule in the specification.

## 5 Conclusion and Future Work

We presented an approach to make abstract system specifications formulated as ASM rules using algebraic data types executable and to provide a visualization of the system state. CoreASM and Eclipse were used for simulation and visualization. Eclipse is a good basis for our purposes as it comes with a rich framework for graphical user interfaces and data binding facilities. CoreASM integrates well into Eclipse and can be extended easily.

This approach has been useful in our work towards the specification and verification of a rather abstract model of UBIFS [SSHR09]. We are currently working on the specification of further layers, that refine the present model and are based on ASMs as well. We plan to integrate a unit test framework like JUnit for automated testing of the specification by executing predefined sequences of operations. Also, the presented approach will allow us to execute the different specifications in parallel, showing the correspondence of operations and data structures of different levels of the UBIFS model.

## References

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[Anl00] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machine Language. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, 2000.

[BS03] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[DBA08] K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In *Proc. of the 10th Int. Conf. on Formal Methods and Sw. Eng. (ICFEM)*, pages 25–44. Springer LNCS 5256, 2008.

[FGG07] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae*, 77(1–2):71–103, 2007.

[Gro03] Microsoft FSE Group. The Abstract State Machine Language. `http://research.microsoft.com/en-us/projects/asml/`, 2003.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford Univ. Press, 1995.

[HL09] W.H. Hesselink and M.I. Lali. Formalizing a Hierarchical File System. *Proc.REFINE 2009, Electronic Notes in Theoretical Computer Science*, 259:67–95, 2009.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[Hun08] A. Hunter. A Brief Introduction to the Design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[JH07] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.

[MS87] Carrol Morgan and Bernard Sufrin. Specification of the UNIX filing system. In *Specification case studies*, pages 91–140. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1987. ISBN: 0-13-826579-8.

[RN05] G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199, March 2005.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured Specifications and Interactive Proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[Sch08] Joachim Schmid. *AsmGofer*. PhD thesis, 2008. Available electronically at `http://www.tydo.de/doktorarbeit.html`.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[SSHR09] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proceedings of FM 2009: Formal Methods*, pages 190–206. Springer LNCS 5850, 2009.