Mechanising a Formal Model of Flash Memory

Andrew Butterfield [a], Leo Freitas [b], Jim Woodcock [b]

[a] *Trinity College Dublin, Ireland*

[b] *University of York, UK*

Abstract

We present second steps in the construction of formal models of NAND flash memory, based on a recently emerged open standard for such devices. The model is intended as a key part of a pilot project to develop a verified file store system based on flash memory. The project was proposed by Joshi and Holzmann as a contribution to the *Grand Challenge in Verified Software*, and involves constructing a highly assured flash file store for use in space-flight missions. The model is at a level of abstraction that captures the internal architecture of NAND flash devices. In this paper, we focus on mechanising the state model and its initialisation operation, where most of the conceptual complexity resides.

Key words: Grand challenge, verification, flash hardware, theorem proving

1. Introduction

The “Grand Challenge in Computing” (Hoare, 2003) (GC) on Verified Software (Woodcock, 2006), has a stream focussing on mission-critical file stores, such as may be required for space-probe missions (Joshi and Holzmann, 2005). Of particular interest are file stores based on the relatively recent NAND flash memory technology, now very popular in portable data storage devices such as MP3 players and datakeys. Flash memory is seen as ideal for these kinds of missions as it has good physical handling properties, being non-volatile, shock-resistant and capable of operating under a wide range of pressures and temperatures. It also has the very valuable property, for space-borne vehicles, of having no moving, and in particular, no rotating parts.

The data in flash memory is structured into pages that are then grouped into blocks, generally with some higher levels of grouping in addition. There are two types of flash memory: (i) NOR flash memory, which can be programmed (written) at byte level, but must be erased at block level, is relatively slow, but suits random access; and (ii) NAND flash memory with higher speed, where programming must be done at the page level, making it a sequential access device. The former suits non-volatile core-memory, whilst the latter is suited to implementing data-stores and file-systems.

This paper is a follow-up to (Butterfield and Woodcock, 2007), which described an initial formal model of NAND flash memory, based on the recently released specification from the “Open NAND Flash Interface (ONFI)” consortium (Hynix Semiconductor et al., 2006). That paper focussed on the structural aspects of the devices, *i.e*, their internal organisation, and an abstract view of the behaviour of those operations

Email addresses: `Andrew.Butterfield@cs.tcd.ie` (Andrew Butterfield), `leo@cs.york.ac.uk` (Leo Freitas), `Jim.Woodcock@cs.york.ac.uk` (Jim Woodcock).

deemed mandatory by the standard. The model presented, using the Z notation (Spivey, 1992; Woodcock and Davies, 1996), was hand-crafted, and intended for human consumption.

The Posix file store pilot project is more complex than the first pilot project, the correctness of the Mondex smart card Woodcock and Freitas (2006). The top-level specification is rather larger, and the amount of code will be much greater (Mondex has been implemented using about 700 lines of JavaCard code). Flash memory is just one component in the Posix project. The others include the Posix interface itself, the refinement through several layers to programming data structures, principally hash maps and search trees, and the memory infrastructure, including device drivers, the flash translation layer, and the model of flash memory itself. The flash memory model is at least as complex as the Mondex problem.

Nevertheless, a key goal of the grand challenge in verification is to develop a repository of mechanically verified software and hardware, and so to this end we proceeded to mechanise the model, using the Z/Eves theorem prover (Saaltink, 2004). This paper describes part of the mechanised model, placing emphasis on how the model had to be elaborated in order to facilitate mechanisation. Here we describe the reformulation of the schemas describing these devices as well as the process of marking defective blocks in the device.

It should also be noted that the model presented in this paper is not an abstract specification of a system to be developed, with a view to avoiding "implementation bias", in order to allow a developer freedom to seek the best solution. Instead, we are modelling an existing artefact (the ONFI specification), and the real devices that already exist or that are likely to come into existence in the near future. As a consequence there will be clear examples of implementation bias in this model.

This is the first work on modelling flash memory, to the extent of our knowledge. The file system is implemented using the interface to flash memory. A formal model of flash memory is required to give semantics to this interface. For example, the correctness of flash memory device drivers relies on a precise understanding of how the memory actually works. There are many different algorithms used to manage flash memory, including its mandatory command set, workload-related aging, wear-levelling algorithms, and memory reclamation (garbage collection). Some of the algorithms involved are intricate and their correctness is not obvious. This is the first step in constructing a more general domain model of memory hardware, including new technologies, such as multi-layer flash and phase-change memory.

We briefly describe related work in §2, and then in §4 we describe the internal organisation of NAND flash devices, while in §5 we discuss the modelling of the initial state of these devices. We finish with a discission of the future progression of this work in §6.

2. Related Work

There has been a considerable body of work done on formal models of file systems, and the technical, usage and reliability aspects of NAND flash devices, but there is little published work on the formal modelling of NAND flash devices at present. Of recent interest, however, is the application of model-checking techniques to Flash memory design (Kim et al., 2008) in collaboration with Samsung, one of the world's largest flash memory manufacturers. This looked at verifying a device-driver operation, using model-checking on a model of a small flash device, and following this up with tests on a large real flash device. A key point made in (Kim et al., 2008) is that testing was proving totally inadequate as a verification technique for their software and it was only the introduction of model-checking techniques that allowed the project to complete satisfactorily.

There has been a considerable body of work done on formal models of file systems, and the technical, usage and reliability aspects of NAND flash devices.

Formal aspects of file systems have covered specifications (Morgan and Sufrin, 1987; Meira et al., 1994; Heisel, 1995; Place, 1995) and approaches towards their verification (Arkoudas et al., 2004). Some recent work has also looked at applying model-checking techniques to entire file systems (Yang et al., 2006), with considerable success. The DAISY file system, implemented in about 1200 lines of Java, was used in 2004 as a case-study for a special joint CAV/ISSTA event Freund and Qadeer (2004). Intermediate findings on using Promela/SPIN and Petri-Net/SMART to check part of the Linux file system is reported as a technical report Andy Galloway and Lüttgen (2007). At another extreme, a exercise applying software model-checking to an entire Linux distribution Schwarz et al. (2005) uncovered 108 exploitable bugs, of which 97 were associated with file system vulnerabilities.

There has been a wide range of material published regarding the implementation of file systems on NAND flash memory, most of which utilise some form of log-structuring (Kim and Lee, 1999; Woodhouse, 2001; Manning, 2002; Intel, 2004; Lim and Park, 2006). Of interest to a potential space-borne application are techniques that use NAND flash to implement low-power file caches for mobile devices (Marsh et al., 1994; Kgil and Mudge, 2006). A key feature of these schemes is the need to cope with the accumulation of errors over time, a mechanism which is very well understood (Aritome et al., 1993; Sikora et al., 2006).

3. Z/Eves

The choice of Z for the Posix interface was made for legacy reasons, as the original Morgan & Sufrin specification (Morgan and Sufrin, 1987) was in Z. It seemed most straightforward to stay in one language so as to avoid a semantic gap, leaving a choice of either Z/Eves (Saaltink, 2004) or ProofPower-Z as a theorem prover. Given the experience in using Z/Eves, this was adopted for tool-support.

We give a brief overview of the Z/Eves theorem prover Saaltink (2004). The prover works on a goal predicate, which is transformed by proof commands into a *logically equivalent* goal — hence it effectively supports equational reasoning. This has an impact on how witnesses are used, so for example, let w be a witness for goal $G = \exists x \bullet P(x)$, then applying this witness results in the new goal $P(w) \lor \exists x \bullet P(x)$, rather than just $P(w)$.

Z-Eves applies a high degree of automation in many of its proof steps, usually exploiting various rewrite and reduction techniques, driven by a database of rules. These commands traverse the goal, left-to-right, top-to-bottom, and either transform sub-parts, or extends a *context* contain predicates that can be assumed true(*e.g.* goal $P \Rightarrow Q$ will transform to context P and goal Q).

The rules come from a variety of sources, most notably toolkit libraries supplied with the tool, and from theorems posited and proved by the users themselves. A variety of mechanisms are provided to give the user control over rule usage:

Rule enabling/disabling Theorems and predicates in schemas can be labelled as named rules (keyword `rule`), and marked if so desired as `disabled`. An enabled rule is applied automatically, if applicable, by the various automation steps, whereas the use of a disabled rule has to be explicitly invoked by the user.

Assumption Rules Any rule of a certain form, containing a *trigger*, can be labelled with the keyword `grule`. If a trigger is encountered, the rule, or a suitably modified form of it, is added to the context, or applied to the trigger.

Forward Rules An theorem whose form is an implication ($P \Rightarrow Q$) can be designated a forward rule, using the `frule` keyword. If predicate P is added to the context, this triggers the rule, which also extends the context with Q.

For further details, see the Z/Eves Reference Manual Meisels and Saaltink (1997). In Z paragraphs, we can add rule annotations as special “comments”, (e.g. ⟪ disabled rule dDataElems ⟫).

4. Flash Memory Structure

In this section we present the various layers representing the way NAND flash devices are organised. At the top level, a NAND flash chip, or *Device*, is composed of a number (1–4) of independent cores, each with their own off-chip communication facilities, called *Targets*. Within a target, there are a number (1–4) of *Logical Units (LUNs)* which can process commands concurrently, but which share communication links with the outside world(see Figure 3, p7).

The memory inside a LUN is arranged as a large number (1024–4096) of *Blocks*, each of which is itself a number (32–96) of *Pages*. In addition, a LUN contains a special page called the *Page Register (PR)*, used as a staging post for data transfers, and a single-byte *Status Register (SR)*, used to report progress and failure of memory operations(see Figure 2, p6).

A page is an array of Bytes, conceptually split into two parts: the main page whose size is a power of 2 (1024,2048), plus a spare part typically 16–64 bytes in length(see Figure 1, p5). The purpose of the spare section is to facilitate error detection and correction, as well as to play a role in marking defective blocks.

We now present the Z model of this structure, working from the bottom level upwards.

4.1. *Data Unit*

The basic data unit in a flash memory is either a Byte (8 bits), or a Word (16 bits), depending on the type of device. We are going to abstract away from this detail, and assume a given type called *Data* that denotes the basic information unit. From previous experience in modularising operations in schemas (*i.e.*, the *Mondex* case-study (Woodcock and Freitas, 2006)), we need witnesses in order to discharge existence proofs, and the ONFI model requires at least two distinguishable *Data* values *zeroed* and *erased*. To this end we introduce a given set *Datum*, where *Data* is defined as its non-empty subset, with the added invariant on *Data*, required later on, that at least two elements exist:

$[Datum]$

$$\begin{array}{|l} Data : \mathbb{P}_1\ Datum \\ \hline \langle\!\langle \text{ disabled dDataElems } \rangle\!\rangle \qquad \exists\, d1, d2 : Data \bullet d1 \neq d2 \end{array}$$

We add a type weakening lemma indicating that elements of *Data* are actually *Datum*. This is important to increase proof automation and avoid problems with Z's maximal type inference.

theorem grule gDataIsDatum (§B)
$d \in Data \Rightarrow d \in Datum$

The Z/Eves keyword `grule` is used here to indicate that this is to be used as an assumption rule, so any addition to a proof context asserting membership in *Data* also adds a context assumption regarding *Datum* membership. The notation (§B) indicates that the proof is to be found in Appendix B.

4.2. *Modelling Memory Structure*

The structure of an ONFI-compliant NAND flash device has five levels of hierarchy, but we cannot abstract away from details of this hierarchy because each level captures some boundary of possible behaviour, with implications for how various operations can be interleaved.

A flattened structure, as presented in the ONFI standard, would be quite complex to reason about, therefore, we have chosen to use promotion (Woodcock and Davies, 1996, Chp 13). This gives benefits in the separation of concerns within the proof effort required for operations, particularly the precondition calculations. Also, as many operations over the memory space affect different entities (*i.e.,* operations over pages, blocks, or logical units), it seems a quite natural choice for promotion. A beneficial side-effect is that the specification becomes easier to follow and read.

In most cases, the promotion is relatively trivial, as each layer of the state component has nothing else but the part to be promoted, and hence there is no need to promote to a schema binding (Woodcock and Davies, 1996, p152). Nevertheless, an interesting discovery was that, although the operations are suitable for promotion, one of them requires quite an elaborate definition. In Z, promotion is usually defined when local operations are used pointwise within some global state. Due to the heterogenous nature of one flash memory operation, we have to define what we call *bulk promotion.* That is, we promote local state changes (*e.g.*, at page, block, logical unit, and target levels) into the global flash device at multiple addresses (*e.g.*, all "bad" and all "good" addresses) at once. To the extent of our knowledge, this kind of use of promotion in Z is new.

4.2.1. *Pages*

A page is an array of data items, consisting of a main page, plus some "spare" locations. (Figure 1). This is the basic unit for writing, or programming. The spare locations are designed to assist with error detection and correction. The page size (*pageCount*) must be a power of two [1] , and the column address bits (*colAddrSize*) must be sufficient to address both the main page as well as the spare area. The page count is not a global constant, but is in fact part of the (fixed) state of an ONFI device that characterises it.

[1] memory page sizes are traditionally always a power of two, as this optimises the use of the address bits, whose range is always a power of two.

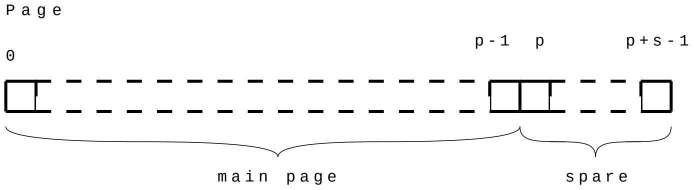

Fig. 1. NAND Flash Structure (Bottom levels)

In Z/Eves we first have to define a function *power* for integer exponentiation, and prove some consistency theorems about the existence of integers related by this function in various ways, as well as useful rewriting rules. Function *power* is defined inductively on the size of the exponent, and we omit it here. Full details about the model and the proofs can be found in Freitas (2007). We can then capture the relationships between page and spare sizes, and the number of bits required to address them in a Z axiomatic definition.

$$
\begin{array}{|l}
pageAddrSize, colAddrSize, pageCount, spare : \mathbb{N}_1 \\
\hline
\langle\!\langle \text{disabled rule dPageCount} \rangle\!\rangle \quad pageCount = power\,(2, pageAddrSize) \\
\langle\!\langle \text{disabled rule dSpareRange} \rangle\!\rangle \quad pageCount + spare \leq power\,(2, colAddrSize)
\end{array}
$$

These rules are marked as disabled to prevent them being applied automatically by the Z/Eves prover, and so have to be explicitly invoked if required. In Z, axiomatic definitions may introduce inconsistencies, and these contradictory axioms may lead to proofs of other inconsistent properties. To avoid this, it is good practice to add consistency theorems for all axiomatic definitions. For instance, before introducing an axiomatic definition, one needs to prove an existential conjecture about the axiom being introduced. If the conjecture is proved, then the axiom is consistent with respect to the preceding specification.

We want to define an axiom on column addresses to represent the addressable space of a page. For this, first we prove an existential conjecture ensuring that column addresses are not empty. This is possible by the definition of the constants above, since they represent strictly positive natural numbers.

theorem tColAddrConsistency (§B)
$\exists\, ca : \mathbb{F}_1\, \mathbb{N} \bullet ca = 0 \mathinner{..} pageCount + spare - 1$

We can now give an axiomatic definition of the (finite) set of all column addresses:

$$
\begin{array}{|l}
colAddr : \mathbb{F}_1\, \mathbb{N} \\
\hline
\langle\!\langle \text{disabled rule dColAddr} \rangle\!\rangle \quad colAddr = 0 \mathinner{..} pageCount + spare - 1
\end{array}
$$

As before for *Data*, to aid with automation, it helps to prove that the column address maximal type is $\mathbb{Z}$:

theorem grule gColAddrMaxType (§B)
$x \in colAddr \Rightarrow x \in \mathbb{Z}$

This makes it easier, for instance, to reason about the use of *power*, defined on $\mathbb{Z}$, when applied to values of type *colAddr*. Now, we define addressable data as a total function from column addresses to data:

$$AddrData \;==\; colAddr \rightarrow Data$$

Finally, we define a *Page* as containing addressable data ranging over the available column addressing space. We use schemas since schema bindings are easier to automate than the usual cross products of various addressing spaces, as used in the original published model (Butterfield and Woodcock, 2007).

$$Page \;\widehat{=}\; [\, info : AddrData \,]$$

At this point we have defined a page as a mapping from column addresses to data items. Similarly, we also have to define a block as a mapping from page addresses to pages, a logical unit (LUN) as having a mapping from block addresses to blocks, a target as having a mapping from LUN addresses to LUNs, and finally (!) a device as a mapping from target identifiers to targets. In each case the relationships between addresses and entities has to be set up with the sequence of axioms and consistency theorems shown above. In the sequel we do not describe all of this but simply concentrate on the parts of Z that differ at the various levels of hierarchy. The complete details, together with proof scripts are available as a technical report (Freitas et al., 2007).

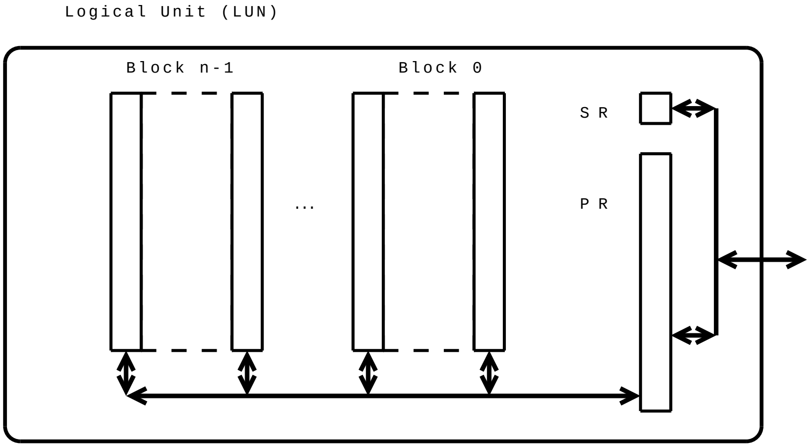

Fig. 2. NAND Flash Structure (Middle levels)

4.2.2. *Blocks*

A block is a collection of pages, and is the smallest unit to which an erase operation can be applied. The number of pages per block is constrained to a multiple of 32.

$$\begin{array}{|ll} \mathit{pagesPerBlock} : \mathbb{N}_1 & \\ \mathit{pageAddr} : \mathbb{F}_1\ \mathbb{N} & \\ \hline \langle\!\langle \text{ disabled dPagesPerBlock } \rangle\!\rangle & \exists\, n : \mathbb{N}_1 \bullet \mathit{pagesPerBlock} = 32 * n \\ \langle\!\langle \text{ disabled rule dPageAddr } \rangle\!\rangle & \mathit{pageAddr} = 0 \mathrel{..} \mathit{pagesPerBlock} - 1 \end{array}$$

$$\mathit{AddrPage} == \mathit{pageAddr} \rightarrow \mathit{AddrData}$$

$$\mathit{Block} \mathrel{\widehat{=}} [\, \mathit{pages} : \mathit{AddrPage} \,]$$

Similarly to *Page*, *Block* is given as a schema with a function on the appropriate addressable space.

4.2.3. *Logical Units*

A logical unit (LUN) is the smallest sub-entity within a device that is capable of operating independently. It comprises a collection of blocks, along with at least one page-register and a status register. The page-registers are used as temporary locations while data is being transferred to and from the LUN. (see Figure 2). For present purposes we assume a single page-register, as ONFI is not specific on this issue (Hynix Semiconductor et al., 2006, p21).

The status register has 8 bits, of which 5 bits currently have defined meanings. Two of these (FAILC, ARDY) are out of the scope of the current model leaving the following three to be considered: *FAIL*, set if a program or erase operation failed; *RDY*, set when the LUN is ready to perform a command; and *WP*, the write-protect flag. Note that the FAIL flag is only valid when RDY is being asserted, while WP is always valid.

$$\mathit{Flag} ::= \mathit{fail} \mid \mathit{ready} \mid \mathit{writeProtected}$$

$$\begin{array}{lcl} \mathit{Status} & == & \mathbb{P}\ \mathit{Flag} \\ \mathit{validStatus} & == & \{\, s : \mathit{Status} \mid \mathit{fail} \in s \Rightarrow \mathit{ready} \in s \} \end{array}$$

The power-up status is that the device is not ready and is write-protected. Once reset (as a result of either power-up or the reset command) is complete, then the status becomes ready and write-protected.

We model LUNs as follows:

$$\begin{array}{|ll} \mathit{blocksPerLUN} : \mathbb{N}_1 & \\ \mathit{blockAddr} : \mathbb{F}_1\ \mathbb{N} & \\ \hline \langle\!\langle \text{ disabled rule dBlockAddr } \rangle\!\rangle & \mathit{blockAddr} = 0 \mathrel{..} \mathit{blocksPerLUN} - 1 \end{array}$$

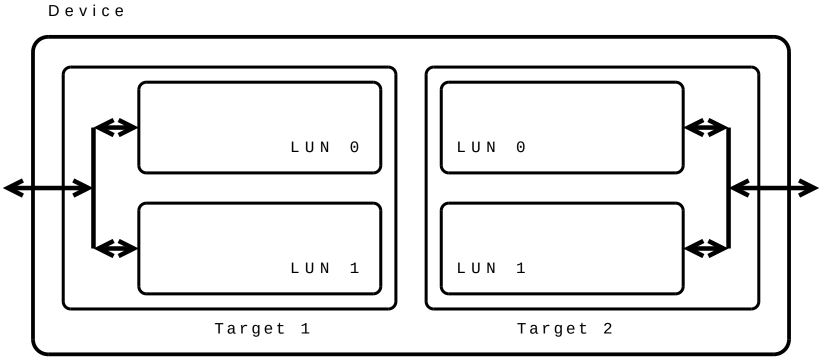

Fig. 3. NAND Flash Structure (Upper levels)

$$AddrBlock \;==\; blockAddr \rightarrow AddrPage$$

$$LUN \;\widehat{=}\; [\, blocks : AddrBlock;\; SR : Status;\; PR : Page \,]$$

LUNs are addressable *blocks* together with the extra page register and status flag.

4.2.4. *Targets*

A target, within a device, is the smallest unit that can communicate independently off-chip. It is made up of one or more logical units.

$$LUNsPerTarget : \mathbb{N}_1$$

$$lunAddr : \mathbb{F}_1\, \mathbb{N}$$

$$\langle\!\langle \text{ disabled rule dLunAddr } \rangle\!\rangle \qquad lunAddr = 0 \mathinner{..} LUNsPerTarget - 1$$

$$AddrLUN \;==\; lunAddr \rightarrow LUN$$

$$Target \;\widehat{=}\; [\, luns : AddrLUN \,]$$

We then view a Target as a map from LUN addresses to LUNs.

4.2.5. *Flash Devices*

A device (single NAND flash chip) encapsulates a number of targets, numbered from 1 upwards. Devices are supplied with a guarantee from the manufacturer regarding an upper bound on the number of bad blocks present. We capture this guarantee as the natural number *maxBadBlocksShipped*.

$$targetsPerDevice : \mathbb{N}_1$$

$$targetIds : \mathbb{F}_1\, \mathbb{N}$$

$$\langle\!\langle \text{ disabled rule dTargetIds } \rangle\!\rangle \qquad targetIds = 1 \mathinner{..} targetsPerDevice$$

$$IdTarget \;==\; targetIds \rightarrow AddrLUN$$

$NANDFlashDevice$

$$targets : IdTarget;$$
$$maxBadBlocksShipped : \mathbb{N}$$
$$badBlocks : targetIds \leftrightarrow (lunAddr \times blockAddr)$$

$$badBlocks \in \mathbb{F}\,(\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z}))$$

$$\#\, badBlocks \leq maxBadBlocksShipped$$

When a device is being used, the application has to maintain a table of bad block addresses for each LUN within all targets. The table size is bounded by the maximum number of blocks, and hence must be finite. As described by ONFI (Hynix Semiconductor et al., 2006, p24), there are algorithms that detect and create this initial table of bad blocks. We abstract from this step and assume this table will be given to the defect marking operation defined below (see Section §5.2). When a device is shipped, it contains an internal representation of the initial state of this table, as the bad blocks are marked as such by the manufacturer.

4.3. *Memory Addressing*

The address data sent into a device conceptually splits into two parts, the row and column addresses. The column address corresponds to an index into a page, while the row address identifies which page is currently being accessed. The row address is itself obtained by concatenating the LUN, block and page addresses in that order. We avoid using tuples as this requires the use of multiple tuple projections when mechanised, so we define equivalent schema representations for the basic address components, and then use schema conjunction to build up the composite addresses:

$$\begin{aligned}
&\mathit{ColAddr} &&\mathrel{\widehat=} [\, ca : \mathit{colAddr} \,]\\
&\mathit{PageAddr} &&\mathrel{\widehat=} [\, pa : \mathit{pageAddr} \,]\\
&\mathit{BlockAddr} &&\mathrel{\widehat=} [\, ba : \mathit{blockAddr} \,]\\
&\mathit{LUNAddr} &&\mathrel{\widehat=} [\, la : \mathit{lunAddr} \,]\\
&\mathit{TargetIds} &&\mathrel{\widehat=} [\, tid : \mathit{targetIds} \,]\\
&\mathit{DataAddr} &&\mathrel{\widehat=} (\mathit{BlockAddr} \land \mathit{PageAddr} \land \mathit{ColAddr})\\
&\mathit{RowAddr} &&\mathrel{\widehat=} (\mathit{LUNAddr} \land \mathit{BlockAddr} \land \mathit{PageAddr})\\
&\mathit{Address} &&\mathrel{\widehat=} (\mathit{RowAddr} \land \mathit{DataAddr})
\end{aligned}$$

The use of *DataAddr* in *Address* rather than the full expanded version as

$$\mathit{Address} \mathrel{\widehat=} (\mathit{RowAddr} \land (\mathit{BlockAddr} \land \mathit{PageAddr} \land \mathit{ColAddr}))$$

is simply to reduce the amount of schema expansion required in the prover — it has no semantic significance.

Proofs involving bulk promotion repeatedly require the proviso of witnesses for certain types of address. Thus, we prove some theorems about the non-emptiness of these address spaces, and that the mappings from addresses to addressed entities are total within the specified addressing spaces.

theorem tExistsColAddr (§B)
$\exists\, ca : \mathit{colAddr} \bullet \mathit{true}$

theorem frule tBlockPagesAreTotal (§B)
$\forall\, \mathit{Block} \bullet \mathit{pages} \in \mathit{pageAddr} \rightarrow \mathit{colAddr} \rightarrow \mathit{Data}$

Here we also indicate, using `frule`, that this theorem should be used as a forward proof rule. This means related schemas do not need to be expanded in order for the prover to infer the fact the theorem establishes. This allows surgical expansion of specific schemas whilst proving more complicated goals, such as precondition calculation or refinement simulations. The result of this is that the list of hypotheses are considerably smaller, hence making the whole proof exercise easier to carry out.

4.4. *Promoting Memory Entities*

We detail below how the framing schemas for the promotion at various levels are laid out. These schemas define how local changes at various levels can be mapped into the global state one or more levels above. That is, how pages are promoted into blocks; blocks into LUNs; LUNs into targets; and targets into devices.

The promotion here is quite subtle, yet trivial. It is subtle because it is not like a normal promotion where sets of bindings are identified, as here we are using promotion to "un-flatten" different addressable spaces. It is trivial because the local states are simple bindings with one component (except for logical units), so we just project the appropriate element, given the right address. As available operations affect specific parts of the state, this layered approach proves to be clear and elegant, yet simple and easy to mechanise. Different concerns are distinctively and modularly separated, whilst proof scripts are quite trivial for the complexity of the data type.

First, we "promote" data within a page given a column address. Although this is more projection rather than promotion, bearing in mind our abstraction from *Byte* and *Word* as fixed-size sequences of bits into *Data*. We could replace *PhiPD* with a more concrete (raw) data type.

$$\begin{array}{|l}
\textit{PhiPD} \\ \hline
\Delta\, \textit{Page};\ \textit{ColAddr}?;\ d! : \textit{Data} \\ \hline
\textit{info}' = \textit{info} \oplus \{\, ca? \mapsto d! \,\} \\ \hline
\end{array}$$

We now prove the following as lemmas:

$$\begin{array}{l}
\textit{info}'\ ca? = d! \\
\{\, ca? \,\} ⩤ \textit{info}' = \{\, ca? \,\} ⩤ \textit{info}
\end{array}$$

That is, the data output ($d!$) at the given column address ($ca?$) comes from the updated pages ($\textit{info}'$), while the data associated with the remaining column addresses within the page remain unchanged. Next, we do the same for pages within a block given a page address. As the *Page* schema only contains one element, we avoid the need to promote into bindings and do it directly into $\textit{info}'$ instead. The before state of *Page* is linked with the corresponding before state of *Block*, since the page at the given address ($pa?$) within the block *pages* is $\textit{pages}\ pa? = \textit{info}$.

$$\begin{array}{|l}
\textit{PhiBP} \\ \hline
\Delta\, \textit{Block};\ \Delta\, \textit{Page};\ \textit{PageAddr}? \\ \hline
\textit{pages}\ pa? = \textit{info} \\
\textit{pages}' = \textit{pages} \oplus \{\, pa? \mapsto \textit{info}' \,\} \\ \hline
\end{array}$$

Again, we promote blocks within a logical unit given a block address, and without referring to *Block* bindings but to $\textit{pages}'$ directly.

$$\begin{array}{|l}
\textit{PhiLB} \\ \hline
\Delta\, \textit{LUN};\ \Delta\, \textit{Block};\ \textit{BlockAddr}? \\ \hline
\textit{blocks}\ ba? = \textit{pages} \\
\textit{blocks}' = \textit{blocks} \oplus \{\, ba? \mapsto \textit{pages}' \,\} \\ \hline
\end{array}$$

Logical units are promoted within a target given a LUN address. This is the only "traditional" promotion (Woodcock and Davies, 1996, Chp.13) in the sense that we are mapping *lunAddr* to *LUN* bindings. That is the case because logical unit schemas also have status and at least one page register, instead of just a function addressing the space where the various levels are laid out.

$$\begin{array}{|l}
\textit{PhiTL} \\ \hline
\Delta\, \textit{Target};\ \Delta\, \textit{LUN};\ \textit{LUNAddr}? \\ \hline
\textit{luns}\ la? = \theta\, \textit{LUN} \\
\textit{luns}' = \textit{luns} \oplus \{\, la? \mapsto \theta\, \textit{LUN}' \,\} \\ \hline
\end{array}$$

We now use the schema version of memory addressing to define data operations over a logical unit of a target:

$$\textit{PhiTargetData}\ \widehat{=}\ (\textit{PhiTL} \land \textit{PhiLB} \land \textit{PhiBP} \land \textit{PhiPD} \land \textit{Address}?)$$

That is, given a full *Address*? as input, which contains column, page, block, LUN, and target addresses, we can perform, in this case, a page-level operation. Similarly, for operations at other levels, we provide varied versions of conjoined schemas representing the right framing with corresponding addressing.

4.5. *Structure Summary*

The model as presented matches very closely the levels of hierarchy described in the ONFI specification, and it may appear that: (i) this is hierarchy for hierarchy's sake; and (ii) this is at too low a level for formal modelling. Nevertheless, each step of the hierarchy captures a distinct change in how the device is accessed and operated, and awareness of these distinctions is important when developing systems where performance is crucial. The ONFI specification also gives descriptions of finite state machines (FSMs) that capture the behaviour of targets and LUNs, viewing these as separate machines communicating with one another. By capturing the target/LUN distinction at this level, we facilitate future work in showing that the FSM view is a refinement of this one. At the level of this model, the only real complexity is the nesting of the various addressing spaces and their mechanisation. This is one example of the implementation bias alluded to in the introduction.

5. Device Initial State

When shipped from the factory, a device will be completely erased (all logic '1's). The only exception to this is for those blocks identified as bad at manufacture time. These blocks will have zeros programmed into specific locations of the spare parts of either their first or last pages. We need to introduce the notion that *Data* has two distinct values *zeroed* and *erased*, among others.

Here there are three instances of *Data*, with *erased* and *zeroed* being different, and this was why *Data* was introduced as having at least two distinct members. Bad block marking is nondeterministic: either the first or the last page address of a block is marked. Also, within the chosen page, any column address in the spare area of a page can be marked, but only one such column address is marked by being zeroed. Therefore, we do not know what the value of the other (not chosen) spare column addresses are. In our original specification, we assumed non-zero values in this case to be erased, but were unable to complete the proofs. The inherent assumption that non-*zeroed* values were in fact *erased* was too strong, and so we had to introduce a third possibility, that the value might be arbitrary and unknown (*any*).

$$\begin{array}{|l}
erased, zeroed, any : Data \\
\hline
\langle\!\langle \text{rule dInitalDataRel} \rangle\!\rangle \qquad \neg\ erased = zeroed
\end{array}$$

For mechanisation, we start by modelling a device with a manufacturer's quality guarantee (*i.e.,* bad block maximum), where the bad-blocks table is left undefined, as we do not yet know what the table is. Subsequently we define a defect marking operation that captures the true state of a shipped NAND flash device.

First we initialise a device with a quality guarantee.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ NANDFlashDeviceInit\ }} \\
NANDFlashDevice' \\
quality? : \mathbb{N} \\
\hline
maxBadBlocksShipped' = quality? \\
\hline
\end{array}$$

We then capture the fact that this guarantee on maximum bad blocks never alters for a given device.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ ShippedNANDFlash\ }} \\
\Delta\ NANDFlashDevice \\
\hline
maxBadBlocksShipped' = maxBadBlocksShipped \\
\hline
\end{array}$$

At this point we can prove precondition theorems for the schemas characterising these defect-free devices.

theorem tNANDFlashDeviceInitPRE (§B)
$\forall\ quality? : \mathbb{N} \bullet \text{pre}\ \ NANDFlashDeviceInit$

theorem tShippedNANDFlashPRE (§B)
$\forall\ NANDFlashDevice \bullet \text{pre}\ \ ShippedNANDFlash$

Finally, targets are promoted within a shipped device (*i.e.,* those initialised devices with maximum bad blocks set).

$$\begin{array}{|l}
\textit{PhiDT} \\
\hline
\textit{ShippedNANDFlash};\ \Delta\ \textit{Target};\ \textit{tid}? : \textit{targetIds} \\
\hline
\textit{targets}\ \textit{tid}? = \textit{luns} \\
\textit{targets}' = \textit{targets} \oplus \{\ \textit{tid}? \mapsto \textit{luns}'\ \} \\
\hline
\end{array}$$

5.1. *Precondition Proof Constants*

A precondition calculates the exact set of before-states in which an operation will re-establish the state invariants in any after-state. This involves a theorem in which the after-state and outputs are existentially quantified, and whose proof therefore requires witnesses to these existential values. We supply these witnesses as theta and lambda terms that return particular instances of interesting data structures, layered according to our definitions above.

$$\begin{array}{lll}
\textit{ANY\_PAGE\_INSTANCE} & == & (\lambda\ \textit{ca} : \textit{colAddr} \bullet \textit{any}) \\
\textit{ERASED\_PAGE\_INSTANCE} & == & (\lambda\ \textit{ca} : \textit{colAddr} \bullet \textit{erased}) \\
\textit{ERASED\_BLOCK\_INSTANCE} & == & (\lambda\ \textit{pa} : \textit{pageAddr} \bullet \textit{ERASED\_PAGE\_INSTANCE}) \\
\textit{ERASED\_LUNBLK\_INSTANCE} & == & (\lambda\ \textit{ba} : \textit{blockAddr} \bullet \\
 & & \quad \textit{ERASED\_BLOCK\_INSTANCE}) \\
\textit{ERASED\_PR\_INSTANCE} & == & \theta \textit{Page}[\textit{info} := \textit{ERASED\_PAGE\_INSTANCE}] \\
\textit{ERASED\_LUN\_INSTANCE} & == & \theta \textit{LUN}[\textit{PR} := \textit{ERASED\_PR\_INSTANCE}, \\
 & & \quad \textit{SR} := \{\ \textit{ready}\ \}, \\
 & & \quad \textit{blocks} := \textit{ERASED\_LUNBLK\_INSTANCE}] \\
\textit{ERASED\_TARGET\_LUNS\_INSTANCE} & == & (\lambda\ \textit{la} : \textit{lunAddr} \bullet \\
 & & \quad \textit{ERASED\_LUN\_INSTANCE}) \\
\textit{ERASED\_TARGET\_INSTANCE} & == & (\lambda\ \textit{tid} : \textit{targetIds} \bullet \\
 & & \quad \textit{ERASED\_TARGET\_LUNS\_INSTANCE})
\end{array}$$

The assignments to θ-expressions above are syntactic sugar in Z/Eves for expression (rather than name) substitution

$$\exists\ i : \textit{AddrData} \mid i = \textit{ERASED\_PAGE\_INSTANCE} \bullet \theta\ \textit{Page}[i/\textit{info}]$$

With these constants, it is now possible to provide witnesses for the specific NAND layers in the various precondition proofs that follow. These constants are just syntactic sugar for a somewhat larger λ-expression representing, say, erased addressable targets (i.e., all LUNS within it are erased, which leads to all blocks and pages to be erased as well).

5.2. *Defect Marking*

The flash memory manufacturing process is not perfect, and so virtually every device shipped will have some bad blocks. The manufacturer tests each device and marks the bad blocks by writing zeros into key locations within the defective blocks. All the good blocks are erased (contents set to all ones).

We model this bad block marking as the operation *NANDFlashMark* whose input is a set of bad block addresses, and whose result is a NAND flash device with those blocks marked as defective, and the complementary blocks marked as erased. This heterogeneous nature of the operation, together with the fact that it updates different parts of the (layered/promoted) states at many points (in bulk) makes this a rather complex operation overall.

Let us first initialise the smallest addressable space: a *Page*. It is defined by the next four schemas below.

5.2.1. *Marking data within a page*

At first, we define a schema used as the signature for the declaration of the *Page* marking operations. This is a useful separation of concerns in case we need to change this signature in the future due to changing requirements. That is, in case of a change in the operation signature, only one place needs to be changed, and hence both the definition of further operations and the related proof scripts will be less affected.

$$PageMarkOp \mathrel{\widehat{=}} [\, \Delta\, Page \,]$$

Usable pages are all those with their addressable space *erased*.

```
┌─ PageMarkErased ──────────────────────────
│ PageMarkOp
├──────────
│ ∀ ColAddr • info′ ca = erased
└───────────────────────────────────────────
```

The bulk promotion is clearly visible in the universal quantifier above. When one needs to prove the precondition of such an operation, this universal quantifier appears within the existential quantifier of the precondition theorem which becomes a hindrance within the proof, which in itself is quite complex. Thanks to the way the constants above were layered, and the fact that automation for λ-expressions is quite high, the (initially intractable) proof becomes much more amenable.

Bad pages are those with at least one *zeroed* data in its spare area (*i.e.,* the column address beyond *pageCount*). Also, it must be a specific page within a block (*i.e.,* either first or last).

Again to separate concerns, and make defining predicates and related theorems more readable, we add constants that pinpoint what the domains are for bad column addresses within a page, and page addresses within a block.

$$\begin{aligned}
&BAD\_COLADDR\_DOMAIN && == \{\, bca : colAddr \mid bca \geq pageCount \,\} \\
&FIRST\_PAGEADDR && == 0 \\
&BAD\_PAGEADDR\_DOMAIN && == \{\, bpa : pageAddr \mid bpa = FIRST\_PAGEADDR \lor \\
& && \qquad\qquad\qquad\qquad bpa = pagesPerBlock - 1 \,\}
\end{aligned}$$

Rules like the ones below are crucial in order to allow automatic proof without expanding the definitions, when abbreviations like the above are used.

theorem grule gBadColAddrDomainMaxType (§B)
$BAD\_COLADDR\_DOMAIN \in \mathbb{P}\, \mathbb{Z}$

theorem rule rBadColAddrDomainElem (§B)
$x \in BAD\_COLADDR\_DOMAIN \Rightarrow x \geq pageCount$

We need weakening rules to keep *BAD_XXXADDR_DOMAIN* disabled without affecting further proofs, e.g:

theorem rule rBadColAddrDomainIsColAddr (§B)
$x \in BAD\_COLADDR\_DOMAIN \Rightarrow x \in colAddr$

It is important that bad addressable domains are not empty, otherwise, it is impossible to mark a particular page as bad, so we prove witness theorems, for example, such as:

theorem rule lPageCountInBadColAddrDomain (§B)
$pageCount \in BAD\_COLADDR\_DOMAIN$

theorem grule gFirstPageAddrMaxType (§B)
$FIRST\_PAGEADDR \in \mathbb{Z}$

theorem rule lBadPageAddrDomainElem (§B)
$FIRST\_PAGEADDR \in BAD\_PAGEADDR\_DOMAIN$

We also state some general theorems, useful in precondition proofs of each stage where we need to provide witnesses, such as:

theorem tExistsBadColAddr (§B)
$\exists \mathit{ColAddr} \bullet ca \in \mathit{BAD\_COLADDR\_DOMAIN}$

theorem tExistsBadPageAddr (§B)
$\exists \mathit{PageAddr} \bullet pa \in \mathit{BAD\_PAGEADDR\_DOMAIN}$

As the first and last pages of a block are where defect marking is done, we actually need a theorem that deals with the pathological case of a block that has only two pages:

theorem tPageAddrDomainBounds (§B)
if $(\mathit{pagesPerBlock} > 2)$ **then**
$\quad(\exists \mathit{PageAddr} \bullet pa \notin \mathit{BAD\_PAGEADDR\_DOMAIN})$
else
$\quad\mathit{pageAddr} = \mathit{BAD\_PAGEADDR\_DOMAIN}$

We can now characterise a marked bad page:

$$\begin{array}{|l}\hline \mathit{PageMarkBad} \\ \mathit{PageMarkOp} \\ \hline \exists \mathit{ColAddr} \mid ca \in \mathit{BAD\_COLADDR\_DOMAIN} \bullet \mathit{info}'\ ca = \mathit{zeroed} \\ \hline \end{array}$$

In order to determine which pages get marked as bad we need a more global view, namely that of the bad blocks within a given LUN. Nevertheless, with the assistance of about six theorems and three further abbreviations, we can prove the precondition:

theorem tPageMarkBadPRE (§B)
$\forall \mathit{Page} \bullet \mathrm{pre}\ \mathit{PageMarkBad}$

The theorems and abbreviations, related to λ-expressions for bad pages/blocks/luns, are similar to those above for erased instances.

5.2.2. *Marking pages within a block*

Once we know how to defect-mark a page, we "promote" it to the defect marking of blocks.

$$\mathit{BlockMarkOp} \mathrel{\widehat{=}} [\Delta \mathit{Block}]$$

We initialise page addresses ($\mathit{PageAddr}$) by marking them accordingly as good or bad and associating such page information with the pages' the block represents. The addressable page ($pa?$) chosen is initialised with $\mathit{PageMarkBad}$ or $\mathit{PageMarkErased}$, accordingly.

$$\mathit{BlockMarkBad} \mathrel{\widehat{=}} (\exists \mathit{PageMarkOp};\ \mathit{PageAddr}? \mid pa? \in \mathit{BAD\_PAGEADDR\_DOMAIN} \bullet \mathit{PageMarkBad} \land \mathit{PhiBP})$$

$$\mathit{BlockMarkErased} \mathrel{\widehat{=}} (\exists \mathit{PageMarkOp};\ \mathit{PageAddr}? \mid \mathit{PageMarkErased} \bullet \mathit{PhiBP})$$

We can prove the preconditions of these schema below

theorem tBlockMarkErasedPRE
$\forall \mathit{Block} \bullet \mathrm{pre}\ \mathit{BlockMarkErased}$

theorem tBlockMarkBadPRE
$\forall \mathit{Block} \bullet \mathrm{pre}\ \mathit{BlockMarkBad}$

5.2.3. *Marking blocks within a logical unit*

A LUN mark operation requires a bulk promotion to take place. Again, bulk here in the sense that more than one point of the local state of pages' for the Block is being updated within the global state of blocks for the LUN, where the points are all those within $\mathit{bas}?$.

$$\begin{array}{|l}
\textit{PhiBulkLB} \\
\hline
\Delta\, LUN;\ Block';\ bas? : \mathbb{P}\ blockAddr \\
\hline
blocks' = blocks \oplus \{\, ba : bas? \bullet (ba, pages') \,\} \\
\hline
\end{array}$$

We must have the format above for constant lambda function automation, as the alternative, $\{\, ba : blockAddr \mid ba \in bas? \bullet ba \mapsto pages' \,\}$, is much harder to prove.

At the LUN level we are finally in a position to provide a input *badBA*? that describes the blocks found to be bad. We need a further invariant stating that good blocks (*i.e.,* all those blocks addressed outside *badBA*?) have no *zero*ed column in any page. That is important, otherwise we could wrongly confuse good blocks with bad. This is an interesting invariant since, although quite obvious, it only appeared during precondition proofs, and is not mentioned in ONFi.

$$\begin{array}{|l}
\textit{LUNMarkSig} \\
\hline
LUN;\ badBA? : \mathbb{P}\ blockAddr \\
\hline
\forall\, DataAddr;\ Block;\ Page \mid \\
\quad pages = blocks\ ba\ \land \\
\quad info = pages\ pa\ \land \\
\quad ba \notin badBA? \bullet \\
\qquad info\ ca \neq zeroed \\
\hline
\end{array}$$

We then define schemas to assist in the description of marking bad blocks in LUNS. It took a degree of experimentation to establish the best schema layer to declare the *badBA*? input. So we we developed a general marking schema, with the declaration signature separate, so keeping changes to a minimum, in particular minimising changes to proof scripts.

$$LUNMarkOp \ \widehat{=}\ [\, \Delta LUN;\ badBA? : \mathbb{P}\ blockAddr \,]$$

We then capture that the page-register is not affected by such bad-block marking, because it is not implemented with the kind of floating-gate technology that is prone to the kind of defect currently under consideration.

$$\begin{array}{|l}
\textit{LUNMark0} \\
\hline
LUNMarkOp \\
\hline
LUNMarkSig \\
(\exists\, PageMarkOp \mid PageMarkErased \bullet PR'.info = info' \land PR.info = info) \\
\hline
\end{array}$$

Finally, blocks in *badBA*? are marked as bad, whilst the rest are erased:

$$\begin{array}{|l}
\textit{LUNMarkBad} \\
\hline
LUNMark0 \\
\hline
(\exists\, BlockMarkOp \bullet BlockMarkBad \land PhiBulkLB[badBA?/bas?]) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{LUNMarkErased} \\
\hline
LUNMark0 \\
\hline
(\exists\, BlockMarkOp \bullet BlockMarkErased \land PhiBulkLB[bas? := blockAddr \setminus badBA?]) \\
\hline
\end{array}$$

The preconditions for these have also been successfully proven, but we omit further details.

5.2.4. *Marking logical units within a target*

To do defect marking at the target level we need to supply a relation *badLBA*? between LUN and block addresses:

$$TargetMarkOp \ \widehat{=}\ [\, \Delta\, Target;\ badLBA? : lunAddr \leftrightarrow blockAddr \,]$$

We can then define the bulk promotion of LUN blocks within a target.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PhiBulkTL}} \\ \hline
\Delta\ \mathit{Target};\ \mathit{LUN}' \\ \hline
\mathit{luns}' = \mathit{luns} \oplus \{\, \mathit{la} : \mathit{lunAddr} \bullet (\mathit{la}, \theta \mathit{LUN}') \,\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TargetMark}} \\ \hline
\mathit{TargetMarkOp} \\ \hline
\mathit{luns}' = \mathit{luns} \oplus (\lambda\, \mathit{la} : \{x : \mathit{lunAddr} \mid x \in \mathrm{dom}\ \mathit{badLBA?}\} \bullet \\
\qquad (\exists\, \mathit{LUNMarkOp} \mid \mathit{badBA?} = \mathit{badLBA?} \llparenthesis\ \{\, \mathit{la} \,\}\ \rrparenthesis \bullet \mathit{LUNMarkBad})) \\
\qquad\qquad \cup \\
\qquad (\lambda\, \mathit{la} : \{x : \mathit{lunAddr} \mid \neg\ x \in \mathrm{dom}\ \mathit{badLBA?}\} \bullet \\
\qquad\qquad (\exists\, \mathit{LUNMarkOp} \mid \mathit{badBA?} = \{\} \bullet \mathit{LUNMarkErased}) \\ \hline
\end{array}$$

Despite the above override covering the whole of *lunAddr*, and hence being a total function, unfortunately this format is not helpful to use lambda abstraction rules about constant functions. Instead, to prove this precondition we need to explicitly show that such set is indeed (partial) functional.

$$\begin{array}{l}
\mathit{luns}' = \mathit{luns} \oplus \{\, \mathit{la} : \mathit{lunAddr};\ \mathit{LUN}' \mid \\
\quad \mathbf{if}\ (\mathit{la} \in \mathrm{dom}\ \mathit{badLBA?})\ \mathbf{then} \\
\qquad (\exists\, \mathit{LUNMarkOp} \mid \mathit{badBA?} = \mathit{badLBA?} \llparenthesis\ \{\, \mathit{la} \,\}\ \rrparenthesis \bullet \mathit{LUNMarkBad}) \\
\quad \mathbf{else} \\
\qquad (\exists\, \mathit{LUNMarkOp} \mid \mathit{badBA?} = \{\} \bullet \mathit{LUNMarkErased}) \,\}
\end{array}$$

This complicates the proof considerably. To avoid that, we rewrite the predicate above as a λ-expression:

$$\mathit{TargetMarkSig}\ \widehat{=}\ [\, \mathit{Target};\ \mathit{badLBA?} : \mathit{lunAddr} \leftrightarrow \mathit{blockAddr} \,]$$

which is similar to TargetMarkOp, but without the after-state

5.2.5. *Marking targets within a NAND flash device*

Finally, we get to the level where we can describe defect marking at the device level, here given an input *badTLBA?* relating target-ids to bad LUN/block address pairs:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NANDFlashMarkOp}} \\ \hline
\mathit{ShippedNANDFlash} \\
\mathit{badTLBA?} : \mathit{targetIds} \leftrightarrow (\mathit{lunAddr} \times \mathit{blockAddr}) \\ \hline
\mathit{badTLBA?} \in \mathbb{F}\,(\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})) \\ \hline
\end{array}$$

and the corresponding bulk promotion:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PhiBulkDT}} \\ \hline
\mathit{ShippedNANDFlash};\ \mathit{Target}' \\ \hline
\mathit{targets}' = \mathit{targets} \oplus \{\, \mathit{tid} : \mathit{targetIds} \bullet (\mathit{tid}, \mathit{luns}') \,\} \\ \hline
\end{array}$$

As the bad blocks within the *ShippedNANDFlash* device are finite due to its limit on the number of allowed bad blocks, we also need to add the constraint that our table of bad blocks is finite. We define the signature as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NANDFlashMarkSig}} \\ \hline
\mathit{NANDFlashDevice} \\
\mathit{badTLBA?} : \mathit{targetIds} \leftrightarrow (\mathit{lunAddr} \times \mathit{blockAddr}) \\ \hline
\mathit{badTLBA?} \in \mathbb{F}\,(\mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})) \\
\#\, \mathit{badTLBA?} \leq \mathit{maxBadBlocksShipped} \\ \hline
\end{array}$$

and the defect marking operation itself:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NANDFlashMark}} \\
\mathit{NANDFlashMarkOp} \\
\hline
\mathit{badBlocks}' = \mathit{badTLBA}? \\
\forall\, \mathit{TargetIds}? \bullet \\
\quad (\exists\, \mathit{TargetMarkOp} \mid \mathit{badLBA}? = \mathit{badTLBA}?(\!|\, \{\, \mathit{tid}? \,\} \,|\!) \bullet \mathit{TargetMark} \land \mathit{PhiDT}) \\
\hline
\end{array}$$

At last, we state the precondition for the device level marking operation:

theorem tNANDFlashMarkPRE
$\forall\, \mathit{NANDFlashMarkSig} \bullet \text{pre}\ \mathit{NANDFlashMark}$

5.3. *Comparison against the original model*

The original model as presented in Butterfield and Woodcock (2007) described the state of a shipped device with bad blocks marked using the following two schemas:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ShipFlash}} \\
\Delta \mathit{NANDFlash} \\
\mathit{quality}? : \mathbb{N} \\
\mathit{badblocks}? : \mathbb{F}(\mathit{TgtId} \times \mathit{LUNAddr} \times \mathit{BlkAddr}) \\
\hline
\#\mathit{badblocks}? \leq \mathit{maxbad}' \\
\mathit{maxbad}' = \mathit{quality}? \\
\forall\, t : \mathit{TgtId};\ \ell : \mathit{LUNAddr};\ b : \mathit{BlkAddr} \bullet \\
\quad \textbf{if } (t, \ell, b) \in \mathit{badblocks}? \\
\quad \textbf{then } \mathit{defectMarked}(\mathit{device}', t, \ell, b) \\
\quad \textbf{else } (\mathit{device}'(t)(\ell)).\mathit{blks}(b) = \mathit{erasedBlk} \\
\hline
\end{array}$$

A defective page is indicated by having a zeroed data item somewhere in the spare area of its first or last page:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{defectMarked}} \\
\mathit{NANDFlash} \\
\hline
\exists\, p : \mathit{PageAddr};\ c : \mathit{ColAddr} \bullet \qquad (p = 0 \lor p = \mathit{pagesperblock} - 1) \\
\quad \land\ c \geq \mathit{pagecount} \\
\quad \land\ (\mathit{dev}(t)(l)).\mathit{blks}(b)(p)(c) = \mathit{zeroed} \\
\hline
\end{array}$$

In this model, entire addresses were built up as products of the various components: row, page, block, LUN and target, and the device was effectively modelled as as a curried mapping from these address components to state values. However, as automation using Z/Eves proceeded,it became clear that this structure, deeply nested in the way that it was, made the proof process very cumbersome, mainly because of the continual need to expand and unpack parts of the model in order to carry out proofs.

The solution, described in this paper, was to extensively re-factor the model, making use of schemas (and particularly schema conjunction) and promotion to capture the hierarchical structure of both the state and the address spaces.

6. Future Work

This is still only early work and there is a lot more to be done. Formal models will be needed to capture the fact that individual targets within a device can be operating concurrently, with interleaving of data-transfers. Also, the behaviour of these devices is described in the specification document using two finite-state machines, one for target behaviour, the other for LUN activity. A model of these needs to be shown as a refinement of the abstract operation model presented in this paper. This requires that the existing operators need to be expressed in terms of basic host/device communication actions, which transfer a single item of information, such as a command, address or data byte/word. The model needs to be extended to cover the non-mandatory operations of the standard, many of which provide improved performance, via various forms of caching and interleaving. It is to be anticipated that any file store will make extensive use of these in order to meet mission performance targets. In many cases a useful model of these will require that the operations are broken down to a smaller granularity.

NAND flash devices are prone to the unrecoverable failure of blocks over time, through what basically amounts to an ageing process, that is strongly workload related. This requires so-called “wear-levelling” algorithms to minimise the failure rate, as well as some form of fault tolerance to cope with the failures that do occur. This requires us to model failure properly, with a particular emphasis on the fact that such failures have a persistent and lasting effect.

We also need to look upwards (in an abstract sense) from the NAND devices to model how they are used to give an illusion of ideal behaviour. Whilst the spare area associated with each data page is there to assist with error detection and recovery, the flash devices themselves have no fault-tolerant mechanisms built-in. Instead the devices have to be interfaced to a controller that manages the faults, and presents a fault-free model of data storage to the level above. A good overview of these issues is Gal and Toledo (2005), which surveys algorithms for flash memory, and will be a key reference for developing models of the file store software levels closest to the hardware.

7. Conclusions

We have described the process of automating part of a hand-crafted Z model of NAND flash memory, using the Z/Eves theorem prover. The mechanised model described here covers the modelling of the NAND flash memory device structure, quite complex in itself, and describing an initialisation operation that characterises the way any real device has bad blocks, scattered at random, but marked by the manufacturer in a certain way. Modelling the patternless nature of this marking proved to be quite a challenge, with the need to explore new forms of promotion.

The key lessons learnt here have been the need to both (i) understand the Z idioms that work best with that prover and (ii) to build up a collection of theorems and lemmas tailored to the proofs that are required, both by the formal methodology involved, as well as the nature and structure of the model. Examples of the former are the use of schema conjunction to describe compound address structures, rather than nested products. For the latter, a series of theorems were required to deal with the issue of maximal types in Z — Z only has the integers as a basic type, so any variable constrained to belong to a defined subset of the integers, ultimately has to be described as being of integer type, with an associated (invariant) predicate.

However, there is an interaction between these two key lessons: the preference for schema conjunction over nested products relies on the following two observations: First, the schema calculus is an integral part of the Z language and methodology, so it is not surprising that it is well supported by the prover. Secondly, the difficulty in using nested product arose largely due to the lack of useful theorems about pairs, triples, pairs of pairs, etc., in the mathematical toolkit supplied with Z/Eves. The issue highlighted by these observations is that the useability of a theorem prover is determined by the collection of pre-packaged theorems and lemma with which it is supplied, plus any extra material added in by users. The key point to take home is that different tools, even if all based on first-order predicate calculus (say), may be quite different in their ability to prove any given theorem.

We draw some more general lessons from all of these observations. The success of mechanising formal models will encourage an approach where models are built hand-in-hand with a tool, rather than following the “handcraft-then-mechanise” approach used here. Modellers will need to be familiar with the idioms best suited to these tools. Also, in building a repository of verified software (with hardware models as well), there will inevitably develop an interest in using different provers/formalisms to model the same thing, or to build a large system from various components, each developed with a different formalism or prover. Even if the various tools have the same or similar underlying logic, and same or similar type systems, the different idioms used for the various tools could present a barrier to success. We anticipate that getting effective interoperability across various mechanised reasoning platforms will require some “standardisation” of the idioms used.

Acknowledgements

We are grateful to Rajeev Joshi and Gerard Holzmann from NASA/JPL who originally suggested the flash file store as a grand challenge pilot project. We received constructive feedback on our formalisation of flash memory from members of IFIP WG 2.3 on Programming Methodology at their meeting in Santa Fe in October 2007. We would also like to acknowledge Amber Huffman of Intel for support within ONFI and for answering many of our technical questions about the standard.

References

Andy Galloway, Jan Tobias Müh;berg, R. S., Lüttgen, G., 2007. Model-checking part of a Linux file system. Technical Report YCS-2007-423, Department of Computer Science, University of York. URL `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2007-423.pdf`

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

Arkoudas, K., Zee, K., Kuncak, V., Rinard, M. C., 2004. Verifying a file system implementation. In: Davies, J., Schulte, W., Barnett, M. (Eds.), ICFEM. Vol. 3308 of Lecture Notes in Computer Science. Springer, pp. 373–390.

Butterfield, A., Woodcock, J., 2007. Formalising flash memory: First steps. In: 12th IEEE International Conference on Engineering Complex Computer Systems, 2007. IEEE Computer Society, pp. 251–260. URL `http://doi.ieeecomputersociety.org/10.1109/ICECCS.2007.23`

Freitas, L., 2007. Crg-12. Tech. rep., University of York, www.cs.york.ac.uk/circus/mc/reports.

Freitas, L., Butterfield, A., Woodcock, J., 2007. Onfi specification in z/eves flash memory hardware specification. Tech. rep., High Integrity Systems Engineering (HISE) Group, University of York. URL `www.cs.york.ac.uk/circus/mc/reports/CRG-12.pdf`

Freund, S. N., Qadeer, S., June 2004. Checking concise specifications for multithreaded software. Journal of Object Technology 3 (6), 81–101, special Issue: ECOOP 2003 Workshop of FTfJP.

Gal, Toledo, 2005. Algorithms and data structures for flash memories. CSURV: Computing Surveys 37.

Heisel, M., 1995. Specification of the unix file system: A comparative case study. In: Algebraic Methodology and Software Technology. pp. 475–488. URL `citeseer.ist.psu.edu/heisel95specification.html`

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

Hynix Semiconductor, et al., 28th December 2006. Open NAND Flash Interface Specification. Tech. Rep. Revision 1.0, ONFI, www.onfi.org.

Intel, October 2004. Intel® Flash File System Core Reference Guide, Version 1. Manual 304436–001, Intel Corporation. URL `sunsite.rediris.es/pub/mirror/intel/flcomp/manuals/30443601.pdf`

Joshi, R., Holzmann, G. J., 2005. A mini challenge: Build a verifiable filesystem. In: Proc. Verified Software: Theories, Tools, Experiments (VSTTE), Zürich.

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

Kim, H., Lee, S., 1999. A new flash memory management for flash storage system. In: COMPSAC. IEEE Computer Society, p. 284. URL `http://computer.org/proceedings/compsac/0368/03680284abs.htm`

Kim, M., Choi, Y., Kim, Y., Kim, H., 2008. Pre-testing flash device driver through model checking techniques. In: ICST. IEEE Computer Society, pp. 475–484. URL `http://doi.ieeecomputersociety.org/10.1109/ICST.2008.55`

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

Manning, C., Sep 2002. Introducing YAFFS, the first NAND-specific flash file system. LinuxDevices.com. URL `http://www.linuxdevices.com/articles/AT9680239525.html`

Marsh, B., Douglis, F., Krishnan, P., 1994. Flash memory file caching for mobile computers. In: Mudge,

T. N., Shriver, B. D. (Eds.), Proceedings of the 27th Annual Hawaii International Conference on System Sciences, Vol. I: Architecture, HICSS'94 (Maui, Hawaii, January 4-7, 1994). Vol. 1. IEEE Computer Society Press, Los Alamitos-Washington-Brussels-Tokyo, pp. 451–460.

Meira, S. L., Cavalcanti, A. L. C., Santos, C. S., 1994. The Unix filing system: A MooZ specification. In: Lano, K., Haughton, H. (Eds.), Object-Oriented Specification Case Studies. The Object-Oriented Series. Prentice-Hall, New York, NY, Ch. 4, pp. 80–109.

Meisels, I., Saaltink, M., 1997. The Z/EVES reference manual (for version 1.5).
URL `http://citeseer.ist.psu.edu/351588.html`

Morgan, C., Sufrin, B., 1987. Specification case studies. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, Ch. Specification of the UNIX filing system, pp. 91–140.

Place, P. R. H., August 1995. POSIX 1003.21 — Real Time Distributed Systems Communication. Tech. rep., Software Engineering Institute, Carnegie Mellon University.
URL `ftp://ftp.sei.cmu.edu/pub/posix/formal/full_spec_2.ps.Z`

Saaltink, M., Jun. 30 2004. The Z/EVES 2.0 user's guide.
URL `http://citeseer.ist.psu.edu/676306.html`

Schwarz, B., Chen, H., Wagner, D., Lin, J., Tu, W., Morrison, G., West, J., 2005. Model checking an entire linux distribution for security violations. In: ACSAC. IEEE Computer Society, pp. 13–22.
URL `http://doi.ieeecomputersociety.org/10.1109/CSAC.2005.39`

Sikora, A., Pesl, F.-P., Unger, W., Paschen, U., 2006. Technologies and reliability of modern embedded flash cells. Microelectronics Reliability 46 (12), 1980–2005.
URL `http://dx.doi.org/10.1016/j.microrel.2006.01.003`

Spivey, J. M., 1992. The Z Notation: A Reference Manual, 2nd Edition. International Series in Computer Science. Prentice Hall.

Woodcock, J., 2006. First steps in the verified software grand challenge. IEEE Computer 39 (10), 57–64.
URL `http://doi.ieeecomputersociety.org/10.1109/MC.2006.340`

Woodcock, J., Davies, J., 1996. Using Z. Intl. Series in Computer Science. Prentice Hall.

Woodcock, J., Freitas, L., 2006. Z/eves and the mondex electronic purse. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (Eds.), ICTAC. Vol. 4281 of Lecture Notes in Computer Science. Springer, pp. 15–34.
URL `http://dx.doi.org/10.1007/11921240_2`

Woodhouse, D., Oct 2001. JFFS: The Journalling Flash File System. Ottawa Linux Symposium 2001.
URL `http://sourceware.org/jffs2/jffs2.pdf`

Yang, J., Twohey, P., Engler, D., Musuvathi, M., 2006. Using model checking to find serious file system errors. ACM Trans. Comput. Syst 24 (4), 393–423.
URL `http://doi.acm.org/10.1145/1189256.1189259`

Appendix A. Exponentiation

We define an auxiliary function used to compute exponentiation of positive natural numbers inductively.

$$power : (\mathbb{N}_1 \times \mathbb{N}) \rightarrow \mathbb{N}_1$$

$$\langle\!\langle \text{disabled rule dPowerBase} \rangle\!\rangle \quad \forall\, b : \mathbb{N}_1 \bullet power\,(b, 0) = 1$$

$$\langle\!\langle \text{disabled rule dPowerInduc} \rangle\!\rangle \quad \forall\, base, exp : \mathbb{N}_1 \bullet power\,(base, exp) = base * power\,(base, (exp - 1))$$

We need to prove this operation is satisfiable:

proof[$power\$domainCheck$]
prove by reduce;
■

The following theorem assists in its use:

theorem disabled rule lPower
$\forall\, base : \mathbb{N}_1;\ exp : \mathbb{N} \bullet power\,(base, exp) =$
if $exp = 0$ **then** 1 **else** $base * power\,(base, (exp - 1))$

We prove its domain is OK:

proof[*lPower*]
with enabled (*dPowerBase*, *dPowerInduc*) *prove by reduce*;
■

We need the above definitions and rules to show that the page count is consistent:

theorem tPageCountConsistency
$\exists\, pCnt, paSize : \mathbb{N}_1 \bullet pCnt = power\,(2, paSize)$

proof[*tPageCountConsistency*]
instantiate *pCnt* == *power* (2, 1), *paSize* == 1;
with enabled (*dPowerInduc*, *dPowerBase*) *prove by rewrite*;
■

We also need it for the spare range consistency theorem:

theorem tSpareRangeConsistency
$\exists\, paSize, caSize, s : \mathbb{N}_1 \bullet power\,(2, paSize) + s \leq power\,(2, caSize)$

proof[*tSpareRangeConsistency*]
instantiate *paSize* == 1, *caSize* == 2, *s* == 1;
prove by rewrite;
with enabled (*dPowerInduc*, *dPowerBase*) *prove by rewrite*;
■

Appendix B. Proofs

We present here a selection of some of the theorems proved, and their proofs.

theorem grule gDataIsDatum
$d \in Data \Rightarrow d \in Datum$

proof[*gDataIsDatum*]
prove by reduce;
■

theorem tColAddrConsistency
$\exists\, ca : \mathbb{F}_1\ \mathbb{N} \bullet ca = 0 \mathrel{..} pageCount + spare - 1$

proof[*tColAddrConsistency*]
prove by rewrite;
apply extensionality;
prove by rewrite;
instantiate *x* == 0;
prove by rewrite;
use pageCount\$declaration;
use spare\$declaration;
*apply inNat*1;
rewrite;
■

theorem grule gColAddrMaxType
$x \in colAddr \Rightarrow x \in \mathbb{Z}$

proof[*gColAddrMaxType*]
prove by reduce;
■

theorem tExistsColAddr
$\exists\, ca : colAddr \bullet true$

proof[*tExistsColAddr*]
instantiate ca $==$ 0;
apply dColAddr;
prove by rewrite;
use pageCount\$declaration;
use spare\$declaration;
*apply inNat*1;
simplify;
■

theorem frule tBlockPagesAreTotal
$\forall\, Block \bullet pages \in pageAddr \rightarrow colAddr \rightarrow Data$

proof[*tBlockPagesAreTotal*]
prove by reduce;
■

theorem tNANDFlashDeviceInitPRE
$\forall\, quality? : \mathbb{N} \bullet \mathrm{pre}\ NANDFlashDeviceInit$

proof[*tNANDFlashDeviceInitPRE*]
prove by reduce;
instantiate targets′ $==$ *ERASED_TARGET_INSTANCE*;
prove by reduce;
with enabled (*ERASED_PAGE_INSTANCE*, *ERASED_BLOCK_INSTANCE*)
    prove by reduce;
■

theorem tShippedNANDFlashPRE
$\forall\, NANDFlashDevice \bullet \mathrm{pre}\ ShippedNANDFlash$

proof[*tShippedNANDFlashPRE*]
instantiate targets′ $==$ *targets*, *badBlocks′* $==$ *badBlocks*,
    maxBadBlocksShipped′ $==$ *maxBadBlocksShipped*;
prove by reduce;
■

theorem grule gBadColAddrDomainMaxType
$BAD\_COLADDR\_DOMAIN \in \mathbb{P}\,\mathbb{Z}$

proof[*gBadColAddrDomainMaxType*]
with enabled (*BAD_COLADDR_DOMAIN*) *prove by reduce*;
■

theorem rule rBadColAddrDomainElem
$x \in BAD\_COLADDR\_DOMAIN \Rightarrow x \geq pageCount$

proof[*rBadColAddrDomainElem*]
with enabled (*BAD_COLADDR_DOMAIN*) *prove by reduce*;
■

theorem rule rBadColAddrDomainIsColAddr
$x \in BAD\_COLADDR\_DOMAIN \Rightarrow x \in colAddr$

proof$[\mathit{rBadColAddrDomainIsColAddr}]$

$\quad \mathit{with\ enabled}\ (\mathit{BAD\_COLADDR\_DOMAIN})\ \mathit{prove\ by\ reduce};$

■

theorem rule lPageCountInBadColAddrDomain

$\quad \mathit{pageCount} \in \mathit{BAD\_COLADDR\_DOMAIN}$

proof$[\mathit{lPageCountInBadColAddrDomain}]$

$\quad \mathit{use\ pageCount\$declaration};$

$\quad \mathit{use\ spare\$declaration};$

$\quad \mathit{apply\ inNat1};$

$\quad \mathit{with\ enabled}\ (\mathit{BAD\_COLADDR\_DOMAIN},\ \mathit{dColAddr})\ \mathit{prove\ by\ reduce};$

■

theorem grule gFirstPageAddrMaxType

$\quad \mathit{FIRST\_PAGEADDR} \in \mathbb{Z}$

proof$[\mathit{gFirstPageAddrMaxType}]$

$\quad \mathit{with\ enabled}\ (\mathit{FIRST\_PAGEADDR})\ \mathit{prove\ by\ reduce};$

■

theorem rule lBadPageAddrDomainElem

$\quad \mathit{FIRST\_PAGEADDR} \in \mathit{BAD\_PAGEADDR\_DOMAIN}$

proof$[\mathit{lBadPageAddrDomainElem}]$

$\quad \mathit{use\ pagesPerBlock\$declaration};$

$\quad \mathit{apply\ inNat1};$

$\quad \mathit{with\ enabled}\ (\mathit{FIRST\_PAGEADDR},\ \mathit{BAD\_PAGEADDR\_DOMAIN},\ \mathit{dPageAddr})$

$\qquad \mathit{prove\ by\ reduce};$

■

theorem tExistsBadColAddr

$\quad \exists\, \mathit{ColAddr} \bullet \mathit{ca} \in \mathit{BAD\_COLADDR\_DOMAIN}$

proof$[\mathit{tExistsBadColAddr}]$

$\quad \mathit{instantiate\ ca}\ ==\ \mathit{pageCount};$

$\quad \mathit{prove\ by\ reduce};$

■

theorem tExistsBadPageAddr

$\quad \exists\, \mathit{PageAddr} \bullet \mathit{pa} \in \mathit{BAD\_PAGEADDR\_DOMAIN}$

proof$[\mathit{tExistsBadPageAddr}]$

$\quad \mathit{instantiate\ pa}\ ==\ \mathit{FIRST\_PAGEADDR};$

$\quad \mathit{prove\ by\ reduce};$

■

theorem tPageAddrDomainBounds

$\quad$ **if** $(\mathit{pagesPerBlock} > 2)$ **then**

$\qquad (\exists\, \mathit{PageAddr} \bullet \mathit{pa} \notin \mathit{BAD\_PAGEADDR\_DOMAIN})$

$\quad$ **else**

$\qquad \mathit{pageAddr} = \mathit{BAD\_PAGEADDR\_DOMAIN}$

proof[*tPageAddrDomainBounds*]

instantiate pa == 1;
with enabled (*BAD_PAGEADDR_DOMAIN*, *FIRST_PAGEADDR*, *dPageAddr*)
  prove by reduce;
use pagesPerBlock$declaration;
*apply inNat*1;
rewrite;
apply extensionality;
prove by rewrite;

■

theorem tPageMarkBadPRE

$\forall \mathit{Page} \bullet \mathrm{pre}\ \mathit{PageMarkBad}$

proof[*tPageMarkBadPRE*]

instantiate info′ == *BAD_PAGE_INSTANCE*;
prove by reduce;
instantiate ca == *pageCount*;
prove by reduce;

■