Concurrent Models of Flash Memory Device Behaviour[*]

Andrew Butterfield and Art Ó Catháin

School of Computer Science & Statistics
Trinity College Dublin
Rep. of Ireland
Andrew.Butterfield@cs.tcd.ie

Abstract. We present a CSP model of the internal behaviour of Flash Memory, based on its specification by the Open Nand-Flash Interface (ONFi) consortium. This contributes directly to the low-level modelling of the data-storage technology that is the target of the POSIX filestore mini-challenge. The key objective was to ensure that the internal behaviour was well-specified, and that it was consistent with the specification of the external interface of such devices. The FDR toolkit was used to perform the revelent refinement/model-checking. In addition to uncovering errors and possible sources of misinterpretation in the ONFi standard, this work also describes a methodology for model data-entry based on a "state-chart" dialect of XML (SCXML) using XSLT to translate into CSP, and HTML, to support validation.

1 Introduction

The "Grand Challenge in Computing" [Hoa03] on Verified Software [Woo06, HLMS07], has a stream focussing on mission-critical filestores, as required, for example, in space-probe missions [JH05]. Of particular interest are filestores based on the relatively recent NAND Flash Memory technology, now very popular in portable datastorage devices such as MP3 players and datakeys.

This paper follows on from initial formal models of NAND Flash Memory, reported in [BW07, BFW09] based on the specification published by the "Open NAND Flash Interface (ONFi)" consortium [$\mathrm{H}^{+}06$]. That work looked at a formal model of flash memory in terms of its internal data storage architecture, and the top-level operations that manipulate that storage.

Here we report work on modelling and analysing the finite-state machines in [$\mathrm{H}^{+}06$] that describe the internal behaviour of flash devices. The modelling was done using machine-readable CSP (CSP_M) [Ros97] and the FDR2 tool [For05] for the analysis, and was reported in detail in an M.Sc dissertation [Cat08]. The emphasis of our flash memory modelling to date has been to focus on the flash memory chips themselves, both their external interfaces as well as their internal behaviour and to interrelate the two. Whilst of interest to the ONFi consortium,

[*] work reported in this paper was partially supported by Science Foundation Ireland

this work has a relevance to the broader community as using an ONFi device is not simply a matter or sequencing top-level atomic operations — in fact few of the operations are atomic, and most are designed to be interleaved, to exploit internal concurrency in the devices to improve performance. Indeed, depending on the hardware configuration, key operations like reading and writing may require interleaving with status checking operations in order to function at all.

In the next section (§2) we describe the relevant aspects of ONFi flash devices, and look at related work (§3). We then proceed to present the development of the CSP model (§4) the analyses performed with it (§5), and conclude (§6).

2 Background

There are two types of Flash Memory: (i) NOR flash, which can be programmed (written) at byte level, but suits random access; and (ii) NAND flash with higher speed and density, but where programming must be done at the page level, making it a sequential access device. The ONFi standard, and this paper, is solely concerned with NAND flash.

A flash memory device is best viewed as a hierarchy of nested arrays of bytes[1], plus additional state and storage facilities at various levels. At the bottom we have *pages*, arrays of bytes, which comprise the basic unit for both writing (programming) and reading (operations *PageProgram* and *Read*). The next level up is the *block*, an array of pages, that is the smallest level at which erasure (operation *BlockErase*) can take place. Blocks are aggregated together under the control of a *logical unit (LUN)*, which is the smallest entity capable of independent (concurrent) execution. A LUN also has one or more local registers the same size as a page (*page-registers*), used as temporary storage when transferring data to/from block pages, and a *status register* recording key information about ongoing operations, or those just completed. The status register has 8 bits, of which only bit 6 (a.k.a "SR[6]"), is of interest, used to indicate the ready/busy status of a LUN. LUNs are collected together into *targets*, which have their own means of communication off-chip. A physical flash memory chip (or *device*) may have several targets, depending on the number of available I/O pins. The work reported in this paper focusses on the target level and below, with a particular emphasis on the interactions between LUNs and their containing target.

2.1 Host-Target Communication

Following the ONFi standard [H$^+$06], we use the term *host* to refer to any entity interacting with a flash memory device. Most communication between a host and target is mediated through a single bi-directional byte-wide I/O port, so the hardware interface is essentially serial. Conceptually, four types of transfer take place across this port:

[1] Some Flash devices are organised on "word" (16-bit) lines, but we ignore this detail in this paper.

Command Write (CW) A single byte denoting a command is sent by the host to the target.
Address Write (AW) A byte denoting part of an address is sent to the target.
Data Write (DW) A data-byte is sent to the target.
Data Read (DR) A data-byte is received from the target.

Additional single-bit control pins determine which of the above transfer types are taking place at any given moment. Executing a typical operation involves a series of transfers of the four types listed above, typically with some waiting inbetween. For example, a *Read* operation involves the following (typical) initial series of transfers:

$$CW(\mathit{readOpcode});\ AW(\mathit{addr}_4);\ \ldots;\ AW(\mathit{addr}_0);\ CW(\mathit{confirm})$$

The host has then to wait whilst the addressed data is pulled from the relevant page into the selected LUN's page-register. One way is to poll the target periodically, asking if the LUN is ready, using the *ReadStatus* operation:

$$CW(\mathit{readStatus});\ DR(\mathit{status})$$

Once the status indicates "ready", the data is drawn out, one byte at a time, until the number n of bytes specified in the read operation has been read.

$$DR(\mathit{byte}_0);\ DR(\mathit{byte}_1);\ \ldots;\ DR(\mathit{byte}_n)$$

The ready/busy part of the status can also be read by hardware directly through an output pin, so we distinguish between the "hardware" and "software" approaches to getting status information. The *WriteProtect* operation is also implemented by a single input pin, rather than via a transfer sequence.

2.2 Flash Translation Layers

The hardware/software subsystem that sits on top of unreliable serial-access flash memory and provides the abstraction of reliable parallel-addressable memory is called the *flash translation layer* (FTL). Most of the extant formal modelling of flash memory filesystems (see §3) assumes the existence of (at least) the hardware parts of the FTL. This paper is concerned with what happens *beneath* the FTL, and so we do not consider it further.

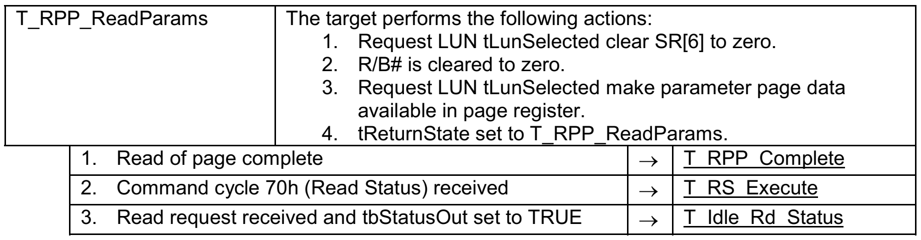
2.3 Flash Memory Operations

The ONFi standard defines a collection of operations that are to be supported by flash devices. Some of the operations are mandatory and must be provided in any ONFi-compliant implementation. The operations, *Read*, *PageProgram*, *BlockErase* and *ReadStatus*, have already been introduced. The other operations include: *Change . . . Column* operations that support access to part of a page; *Reset* to allow software to reset a device,; *WriteProtect* to direct LUNs to be locked/unlocked against changes; and *ReadID* and *ReadParameterPage* that return data specific to a device such as manufacturer's name, and sizing information.

Other optional operations are also specified, typically providing enhanced performance-improving features that exploit the parallelism provided by the LUNs.

2.4 The ONFi state machines

The internal behaviour of ONFi devices is described by two finite-state machines (FSMs) [H^+06, §7], one describing the behaviour of a target, the other capturing the actions of a LUN. The target state machine is defined with the aid of seven state variables, and has a total of 77 state entries. The LUN state machines uses eight state-variables and 62 states. An example state entry, for the target state `T_RPP_ReadParams` (for the *ReadParameterPage* operation) is shown in Fig.1. We shall use this as a running example to describe our approach. The box at on the top-right describes the events that occur on entry to the state. The three rows below describe the subsequent conditional behaviour in this state. The left of each row describes a input event or condition whilst the right indicates the resulting state transition, with the conditions being evaluated in the order in which they appear.



T_RPP_ReadParams	The target performs the following actions: 1. Request LUN tLunSelected clear SR[6] to zero. 2. R/B# is cleared to zero. 3. Request LUN tLunSelected make parameter page data available in page register. 4. tReturnState set to T_RPP_ReadParams.	
1. Read of page complete	→	T_RPP_Complete
2. Command cycle 70h (Read Status) received	→	T_RS_Execute
3. Read request received and tbStatusOut set to TRUE	→	T_Idle_Rd_Status

Fig. 1. ONFi Target State example [H^+06]

3 Related Work

Formal model-checking techniques have been applied to the verification of the Samsung OneNAND flash device driver [KCKK08], with particular emphasis on a multi-sector read operation implemented within the FTL. This proved too complex for "conventional testing methods"[2] to the extent that even when tests failed, they were not adequate to pinpoint the cause of the error. The model-checkers explored were NuSMV, Spin and CBMC. The best tool was reported

[2] Their words

as CBMC[CKL04], a SAT-solver based model-checker, that works directly with C source code. It was able to uncover a number of previously unknown bugs in critical sub-systems of their FTL.

A fully automatic analysis, using Alloy, of a flash filesystem is described in [KJ08]. This was built on top of a simple flash model (at roughly the same level of abstraction as [BW07]). and implements wear-leveling and block mapping, so covering the "soft" parts of the FTL. Similar work, but very much a tools-integration approach to modelling (VDM/HOL/Alloy), is reported in [FSO08]. The key issue here is matching specific tools to specific verification tasks, and the need to translate between tool notations, in order to have a complete formal verification lifecycle. VDM is used as the main modelling tool, with Alloy and HOL called upon to verify proof obligations that arise.

At the other end of the scale, there is ongoing work on the modelling of the filesytem from the POSIX level down. This ranges from explorations of modelling the tree structures characteristic of filesystems (e.g. acyclic graphs), in Event-B using the Rodin platform [DBA08], to comprehensive machine verification of the POSIX Z model [FWF09] and part of the IBM CICS system [FWZ09]. Finally, we note recent work looking at computational models of flash memory devices with performance issues in mind [ABJ^{+}09], of possible interest to the formal verification community as they suggest the kinds of optimisations to be considered during the later stages in the refinement to code.

In terms of automated translations from some notation into CSP, we note the Casper tool developed by Gavin Lowe [Low98], designed for cryptographical protocols — however this used a tailored notation not suitable for our purposes.

4 The CSP Model

The main objective of this work was to formalise the Target/LUN FSM descriptions in machine-readable CSP and then use this as a basis for checking their correctness using the FDR2 refinement checker [For05]. CSP was chosen because of its familiarity, and the availability of the FDR2 model checker, and because the basic mechanisms of CSP appeared to be a good match for the FSM model in the ONFi document.

The main criteria for correctness was that the behaviours possible for the interconnected FSMs was consistent with the behaviour patterns for the operations mandated by that same standard.

The state machine notation of the ONFi specification allows for a relatively direct conversion into CSP: there is a one-to-one mapping between ONFi states and CSP processes. The ONFi FSMs interact by passing messages and waiting to respond to same, dependent on both the named-state they are in and conditions over other state-variables. The conceptual match between this and CSP processes is very close, as examples later will show. The separation of target from LUNs also echoes the parallel composition features of CSP. Multiple LUN processes can be interleaved: required to synchronize on events with the target, but not with each other. The target-LUN communication events (*TLEvts*) are then hidden and this is put in parallel with a *HOST* process that models the behaviour of the environment that communicates with the flash device. In CSP notation this is written (for a single target and two LUNs) as:

$$SYSTEM \mathrel{\widehat=} HOST \parallel ((\,TARGET \parallel (\,LUN(0) \interleave LUN(1)\,)\,) \setminus TLEvts)$$

Modelling the communication between host and target was straightforward as this is well documented as the external interface of ONFi devices, and had already been modelled in Z at an abstract level[BW07, BFW09]. In CSP_M we used events with names of the form `ht_XXXX` to model these communications, which basically consisted of the byte-level transfers of commands, addresses, data and the single-bit signals (e.g. write-protect input, ready/busy output).

Details of the target-LUN communication (CSP_M events of form `tl_XXXX`) were much more sketchy, precisely because these are viewed as implementation details to be resolved appropriately by individual device manufacturers. For example, during a *PageProgram* operation, the specification goes into some detail during the input of address bytes from the host to the target. For the transfer of the same address from the target to the appropriate LUN, it simply states "Target issues the [page] Program with associated row address to the LUN" [H^+06, p84]. It is assumed that the target can transfer the address in one go, rather than serially, byte-by-byte.

Certain abstractions and simplifications had to be made so that the FDR2 model-checker could perform analysis without running out of memory. So, as just seen above, most data and address items were modelled as single bits, while the command datatype was restricted to the set of known command types, rather than being a full byte. An exception is the *column address* (address of byte within page), which was modelled as two bits to support the *ChangeXXXXColumn* operations.

The 7 state variables of the target FSM had also to be abstracted, and augmented with implicit state data, such as the state of the write protect pin, and the data and address information temporarily in transit, as well as a counter for the number of address chunks expected. This resulted in the addition of a further 12 state components. A similar exercise in augmenting the state had to be done for the LUN FSM as well, to a lesser degree (8 ONFi variables were augmented by a further 3).

4.1 CSP Data-Entry: a challenge

Generating the CSP models for *TARGET* and *LUN* was a considerable challenge, best illustrated by considering the CSP encoding in Fig.2 of the state shown previously in Fig.1, where we explicitly list the 19 variables needed. A typical state transition is triggered by a condition on a small subset of those state-variables, and itself usually only modifies a few of them. Clearly the tasks of both entering the data for, and checking the correct encoding of, each of the state-tables, was a daunting and highly error-prone task.

An additional complication arose from the fact that textual ordering is used to determine which state transitions occur if more than one is possible. So given

```
T_RPP_READPARAMS(tbStatusOut,tbChgCol,tCopyback,tLunSelected,tLastCmd,tReturnState,
     tbStatus78hReq,cmd,isReadyBusy,isWriteProtected,dataBit,addrReceived,lun0ready,
     lun1ready,intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL) =
         tl.tLunSelected!targRequest -> tl_setSR6.tLunSelected!false ->
         tl.tLunSelected!targRequest -> tl.tLunSelected!retrieveParameters ->
                 (tl.tLunSelected.readPageComplete -> T_RPP_COMPLETE(tbStatusOut,
                     tbChgCol,tCopyback,tLunSelected,tLastCmd,T_RPP_ReadParams,
                     tbStatus78hReq,cmd,false,isWriteProtected,dataBit,addrReceived,
                     lun0ready,lun1ready,intCounter,addr3Block,addr2Page,addr1ColH,
                     addr0ColL)
                     []
                 ht_ioCmd.cmd70h -> T_RS_EXECUTE(tbStatusOut,tbChgCol,tCopyback,
                     tLunSelected,tLastCmd,T_RPP_ReadParams,tbStatus78hReq,cmd,false,
                     isWriteProtected,dataBit,addrReceived,lun0ready,lun1ready,
                     intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL)
                     []
                  (tbStatusOut==true)
                      & (ht_read -> T_IDLE_RD_STATUS(tbStatusOut,tbChgCol,tCopyback,
                     tLunSelected,tLastCmd,T_RPP_ReadParams,tbStatus78hReq,cmd,false,
                     isWriteProtected,dataBit,addrReceived,lun0ready,lun1ready,
                     intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL)))
```

Fig. 2. CSP encoding

an ONFi table of the form on the left, we had to generate CSP_M in the form on the right:

c1 and e1	→	S1
c2 and e2	→	S2
c3 and e3	→	S3

```
c1                               & e1 -> S1
(not c1) and c2                  & e2 -> S2
(not c1 and not c2) and c3 & e3 -> S3
```

After some initial experimentation with small handcrafted examples, it became very clear that some form of automation was going to be needed if the CSP encoding was going to be completed in a timely fashion. The solution adopted was to use State Chart XML (SCXML), a "state-chart" dialect of XML [BAA$^+$09] for initial data entry. This was chosen because SCXML provided a textual way to describe states, state-variables, and variable updates at a level very close to that used in the ONFi descriptions. Given an SCXML encoding, this could then be translated into the machine-readable form of CSP using XSL Transformations (XSLT) [W3C99]. The ready availability of parsers and tools to manipulate XML made this an easier prospect than trying to develop our own data-entry language with tool support.

The SCXML code to describe the `T_RPP_ReadParam` state is shown in Fig.3. The key feature to note is that the data-entry requirements are limited to the information that appears explicitly in the ONFi behaviour tables.

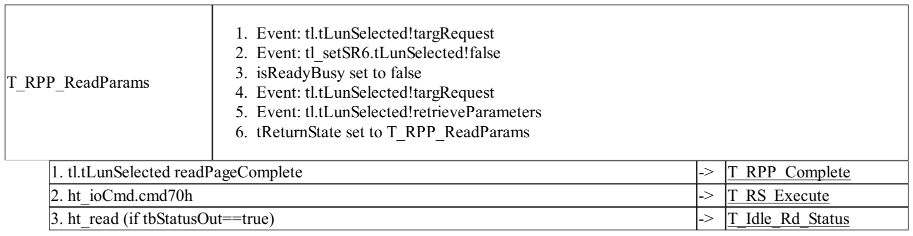
One caveat has to be mentioned: the SCXML we used can be processed by the standard XSLT translation tools, but is not itself correct SCXML. We are using the `<event name="...">` construct, but our 'names' are in fact portions of CSP_M event syntax. However, as we are simply using SCXML for data-entry to build a CSP model, the fact that we cannot use SCXML tools for analysis is not a concern.

Validating the data entry is important, and is facilitated by the fact that these same SCXML sources can also be used to generate HTML that renders

```
<state id=T_RPP_ReadParams>
    <onentry>
        <event name=tl.tLunSelected.setSR6!0"/>
        <assign location=readyBusy expr=0"/>
        <event name="tl.tLunSelected!retrieveParameters"/>
        <assign location="tReturnState" expr="T_RPP_ReadParams"/>
    </onentry>
    <transition event=tl.readPageComplete target=T_RPP_Complete/>
    <transition event=ht_Iocmd.cmd70h target=T_RS_Execute/>
    <transition cond=tbStatusOut == true
                event=ht_read target=T_Idle_Rd_Status/>
```

Fig. 3. SCXML encoding

in a style very close to that used by ONFi (Fig.4) — this greatly facilitates the checking and proof-reading of the entered data. The difference in the number of state-entry events (6 rather than 4) is that single events in the ONFi document are sometimes split into several at the SCXML/CSP level.



T_RPP_ReadParams	1. Event: tl.tLunSelected!targRequest 2. Event: tl_setSR6.tLunSelected!false 3. isReadyBusy set to false 4. Event: tl.tLunSelected!targRequest 5. Event: tl.tLunSelected!retrieveParameters 6. tReturnState set to T_RPP_ReadParams	
1. tl.tLunSelected readPageComplete	->	T_RPP_Complete
2. ht_ioCmd.cmd70h	->	T_RS_Execute
3. ht_read (if tbStatusOut==true)	->	T_Idle_Rd_Status

Fig. 4. HTML rendering

5 Model Analysis

The model analysis fell conceptually into two phases: the first focussed on debugging and validating the model, to ensure that it captured the intent of the ONFi specification. The second phase was using the model to analyse the consistency of the entire ONFi document.

5.1 Validating the Model

To model the behaviour of a flash device fully, several processes were necessary in addition to HOST, TARGET, and LUN. These processes were simpler than those derived directly from the ONFi state machine, and so were written in CSP directly rather than via SCXML. The need for these emerged as the model was being built and animated, using the CSP_M ProBE tool.

The first difficulty arose in trying to model the propagation of status information from the LUNs, via the target, to the host. In the ONFi document these are handled within the FSM framework, as events between the LUNs and target. A particular problem arose in relation to bit 6 of this register ("SR[6]"), used to record the "Ready/Busy" status of the system (Ready=1,Busy=0). The SR[6] values of the LUNs are propagated by the target to the host via a single bit pin called "R/B#". The ONFi document states (p19) that

> " R/B# shall reflect the logical AND of the SR[6] (Status Register bit 6) values for all LUNs on the corresponding target. "

In effect the propagation of SR[6] from LUNs to target occurs asynchronously, and concurrently with any other FSM behaviour — trying to model this behaviour exactly as described in the ONFi FSM descriptions led to a deadlocking model. Attempting to augment the target model to sample the SR[6] events more often did not resolve the deadlock in all cases, and so in order to get a deadlock-free model that captured the behaviour intended by ONFi, we had to model the SR[6] and R/B# bit communication using a separate process $READYBUSY$. This allowed the asynchronous updating of SR[6] and hence R/B# to be decoupled from the target FSM, and made available directly to the host.

However, there were still circumstances that required the target itself to be aware of changes in any SR[6] values, particularly where interleaved operations were concerned. These situations essentially arose when the target was itself in an idle state, so both the target and $READBUSY$ processes had to be augmented to communicate with each other at such times. The final architecure of the CSP model now consisted of the main processes and linkages shown in Fig. 5.

Whilst the bulk of the behaviour of the combined FSMs was deterministic, there was one area of unpredictability that we modelled with non-deterministic choice. This was related to the fact that the time it took for certain operations (*Read*,*PageProgram*) to complete was variable, depending on how much data was being processed. We used a process called $LUN\_INNARDS$ to model this, using a counter to record the amount of remaining work, and non-deterministic choice to decide on how much work (always non-zero) was done in one "step". The effect of this was ensure that a bounded number of status reads would return busy, before finally switching to ready.

5.2 Verifying the FSMs

The combination of Target and LUNs was not deadlock-free: they model passive hardware that is meant to be driven by an active host, and so if left to run freely together they quickly enter inconsistent states. So, our analysis had to consist of an appropriately chosen collection of hosts. We came up with two types of analysis: those where the host followed the device usage protocols described in the ONFi standard, which we expected to be deadlock- and livelock-free,

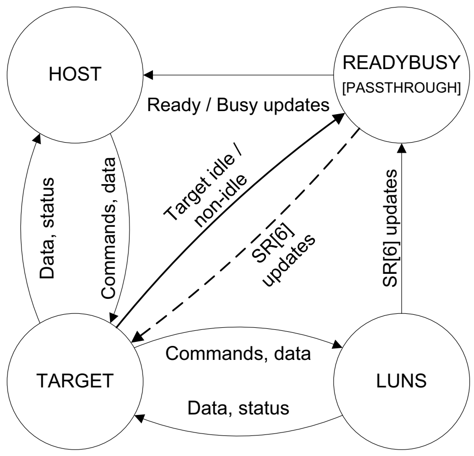


Fig. 5. Final CSP Process Structure

and those where we deliberately modelled incorrect host behaviour, hoping to see deadlocks. Deadlock freedom ensured that the protocols were correct, in so far that both the target and LUN FSMs could follow through the required sequence of events. For deadlock checking we had to ensure that host itself did not terminate, so a typical test host was something that repeatedly performed arbitrary commands. Livelock freedom was checked in the case were all but target-host events were hidden, so ensuring that the host would never run the risk of having to wait forever for the target to complete some operation.

We used two host models — on assuming the hardware approach to status checking, the other that it would be done in software (i.e. explicit *ReadStatus* operations). Either type of host was implemented as an infinite loop that made a non-deterministic choice of an operation to execute on each iteration. The host would then be placed in parallel with a target and two LUNs.

```
TARGET_TWOLUNS = TARGET [| tl_events |] (LUN(lun0) ||| LUN(lun1))
HOST_SW_TARGET_TWOLUNS
           = INITIAL_HS_POWERON [| ht_sw_events |] TARGET_TWOLUNS
```

A key issue that arose early on was that some of the models were too large to even compile in FDR ("failed to compile ISM"), let alone model-check. These were the models that covered all the behaviour in the ONFi FSMs, including that for both the mandatory and optional operations. One of the FDR compression techniques, "`diamond`", was used, as was increasing the stack size (Unix command `ulimit -s 262144`), but in order to get results, the SCXML and XSLT sources were configured in such a way that a subset of the models containing only the states and transitions for the mandatory operators could be automatically generated — it was these `-mandatory` CSP_M files that were used for automated analysis. The one exception to this was that we included the non-mandatory operation *ReadStatusEnhanced*, as this was required in order to test the concurrent operation of two LUNs.

When checking `HOST_SW_TARGET_TWOLUNS` for deadlock freedom FDR2 reported 4490300 transitions during compilation, that it refined 32,338 states with 78469 transitions, and took 17mins to run on a dual 1.28Ghz UltraSPARC-IIIi processor with 4Gb RAM, and 20G swap disk.

For example, a test of the Read operation was set up as follows. We took the `HOST_SW_TARGET_TWOLUNS` process, hid all events except the host-target read-related commands and data transfers, and treated this as a specification.

```
READ_SPEC = HOST_SW_TARGET_TWOLUNS
              \ diff(Events,
                  union({ht_ioCmd.cmds
                         | cmds <-{cmd30h,cmd00h,cmd70h,cmdFFh}}
                       ,{|ht_ioDataOut|}))
```

We then defined a process that performed an expected sequences of host target protocol events for a *Read* (preceded by a *POWERON* behaviour, as it is present in the specification model),

```
POWERON
  = ht_ioCmd.cmdFFh -> ht_ioCmd.cmd70h -> ht_ioDataOut.true -> SKIP
READ_IMPL
  = POWERON;
    ht_ioCmd.cmd00h -> ht_ioCmd.cmd30h
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.false -- status busy, so wait
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true -- read ready, so read
    -> ht_ioCmd.cmd00h -> ht_ioDataOut.false
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true -> STOP
```

We then used FDR to check for trace-refinement.

```
assert READ_SPEC [T= READ_IMPL
```

This check took 47mins on the dual 1.2GHz 4Gb 20Gb swap Sparc machine already mentioned.

We also tested for illegal usage of the device, looking at erroneous actions like the following: *BlockErase* followed by *ReadStatusEnhanced*; *Read* completed without *ReadStatus* (in software model); and Multiple *Read*s completed followed by busy indicator. Most of these came from gleaning the ONFi document for statements regarding required patterns of behaviour.

5.3 Anomalies Uncovered

We have already alluded to the difficulties in how the SR[6] and R/B# pin behaviour, asynchronous in nature, was described in the FSM format, and how we had to model it as a separate process — this was not an error in the ONFi document, but is rather a clarification of what correct behaviour was meant.

Several deadlocks were found the in *ReadParameterPage* operation, one ironically caused by the target requesting a status update just as a LUN decided, unsolicited, to provide such an update. In effect the ONFi standard talks about explicit status update messages when in fact this all occurs asynchronously via bit SR[6]. It was possible to fix this, by adding extra target-LUN synchronisation events (`tl_sync`), but this was now no longer consistent with the implicit requirement that a host can perform a Read Status at any time to determine the device's readiness.

Another deadlock resulted from the user of the `tReturnState` state variable to allow some states to 'return' to a saved 'calling state'. Essentially on return to the saved state, its entry events and state changes get re-executed, involving setup communication with the LUNs, which in certain cases were not expecting it, as they had already been setup.

A number of deadlocks were also found in the interaction between the *Reset* and *ReadStatus* operations.

All of the above were reported back to the ONFi consortium, some of which have lead to an ONFi Technical Errata notice being issued (Errata Id 008). It is worth pointing out that all the deadlock issues we have discovered are connected to the *ReadStatus* operation in some way, and seem to be consequences of trying to mix the asynchronous reporting of status with the synchronised behaviour of the FSMs.

6 Conclusions & Future Work

It is safe to say that to verify a specification as complex as ONFis by hand would have been impossible. Here the one-to-one correspondence between CSP processes and state machine states allowed for a fairly direct conversion, avoiding the need to abstract away too much detail.

Unfortunately the full ONFi model proved too much for the FDR2 model-checker, which failed to compile. The deadlocks described above were discovered in the mandatory-only model. With this limited model we found that the ONFi specification was basically sound, once we had resolved the synchronous/asynchronous mismatch in the description of status reporting. Feedback to ONFi has resulted in corrections to the published specification, now at version 2.1.

Using XSLT to convert the intermediate XML to CSP undoubtedly saved time and allowed a more thorough model to be developed. The conversion is not totally automatic, requiring manual intervention for the following: specification, in CSP, of channels, datatypes and sets to differentiate mandatory from optional commands; minor supplementary CSP processes ($LUN\_INNARDS$, $READYBUSY$); parallel composition of host / target / LUN state machines; and specification of deadlock/livelock checks. The above totalled 583 lines of CSP, whereas the XSLT translations of the full target and LUN models resulted in a total of 1348 lines of CSP. These numbers belie the fact that the generated CSP is far more complex and inscrutable than the hand-generated material.

6.1 Future Work

The full model, including optional commands, remains to be verified. To succeed, some creativity will be required, since the CSP model (as it currently stands) runs into the resource limits of the FDR2 model-checker. One possible approach will be to use FDR Explorer [FW09] to assist us.

ONFi have since released version 2.0 and 2.1 of the specification. The state machine has not changed significantly, so it should be modelled in the same framework without much difficulty, which would also bring in changes due to any relevant errata.

At the time of writing, work is underway to take the SCXML sources and translate them to *Circus*[OCW06], to allow them to be analysed against the top-level Z models we already have. We also propose to use the same sources with new translations to do some comparative work by re-targeting at tools used with other formalisms. Finally, we intend to use an implementation of the "hard" FTL components as a case-study for a ongoing work on hardware compilation.

6.2 Acknowledgments

We'd like to thank Amber Huffman of Intel and Michael Abraham of Micron for their assistance and feedback regarding the ONFi standard, and Micheal Goldsmith of Formal Methods (Europe) Ltd., for his assistance with FDR2.

Sources of the material mentioned in this paper can be downloaded from `https://www.cs.tcd.ie/Andrew.Butterfield/Research/FlashMemory/` .

References

[ABJ$^+$09] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, *SEA*, volume 5526 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2009.

[BAA$^+$09] Jim Barnett, Rahul Akolkar, R. J. Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, and Torbjörn Lager. State chart XML (SCXML): State machine notation for control abstraction. World Wide Web Consortium, Working Draft WD-scxml-20090507, May 2009.

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4):219 – 237, 2009. Special Issue on the Grand Challenge.

[BW07] Andrew Butterfield and Jim Woodcock. Formalising flash memory: First steps. In *ICECCS*, pages 251–260. IEEE Computer Society, 2007.

[Cat08] Art Ó Catháin. Modelling flash memory device behaviour using CSP. Taught M.Sc dissertation, School of Computer Science and Statistics, Trinity College Dublin, 2008. Also published as techreport TCD-CS-2008-47.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[DBA08] Kriangsak Damchoom, Michael Butler, and Jean-Raymond Abrial. Modelling and proof of a tree-structured file system. In *ICFEM 2008*, volume LNCS 5256, pages 25–44. Springer, October 2008. Springer LNCS 5256.

[For05] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement, FDR2 User Manual*, 6th edition, June 2005.

[FSO08] M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying intel flash file system core specification. In P.G. Larsen J.S. Fitzgerald and S. Sahara, editors, *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, pages 54–71, School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.

[FW09] Leo Freitas and Jim Woodcock. FDR explorer. *Formal Asp. Comput*, 21(1-2):133–154, 2009.

[FWF09] Leo Freitas, Jim Woodcock, and Zheng Fu. POSIX file store in Z/eves: An experiment in the verified software repository. *Sci. Comput. Program*, 74(4):238–257, 2009.

[FWZ09] Leo Freitas, Jim Woodcock, and Yichi Zhang. Verifying the CICS file control API with Z/eves: An experiment in the verified software repository. *Sci. Comput. Program*, 74(4):197–218, 2009.

[H$^+$06] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0, ONFI, www.onfi.org, 28th December 2006.

[HLMS07] Tony Hoare, Gary T. Leavens, Jayadev Misra, and Natarajan Shankar. The verified software initiative: A manifesto. http://qpq.csl.sri.com/vsr/manifesto.pdf, 2007.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[JH05] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. In *Proc. Verified Software: Theories, Tools, Experiments (VSTTE), Zürich*, 2005.

[KCKK08] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Pre-testing flash device driver through model checking techniques. In *ICST*, pages 475–484. IEEE Computer Society, 2008.

[KJ08] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in alloy. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[OCW06] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for circus. In *REFINE 2006*, pages 1–16. ENTCS, 2006.

[Ros97] A.W. Roscoe. *The Theory and Practise of Concurrency.* Prentice-Hall (Pearson), 1997. revised to 2000 and lightly revised to 2005.

[W3C99] W3C. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[Woo06] Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.