Modelling Flash Devices with FDR: Progress and Limits*

Arshad Beg
School of Computer Science & Statistics
Trinity College Dublin
Rep. of Ireland
begm@scss.tcd.ie

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

ABSTRACT

We present our experience of working with the Failures-Divergence Refinement (FDR) toolkit while extending our modelling of the behaviour of Flash Memory. This effort is a step towards the low-level modelling of data-storage technology that is the target of the POSIX filestore mini-challenge. The key objective was to advance previous work presented in [4, 2] to cover the full Open Nand-Flash Interface (ONFi) 2.1 model. The previous work covered a sub-model of the mandatory features of ONFi 1.0. The FDR toolkit was used for refinement/model-checking. In addition to the compression techniques available in FDR, we also experimented with FDR Explorer - an application-programming interface (API) that allowed us to get a better picture of FDR performance. This paper summarises the progress we made, and the limits we encountered. We are now able to verify many of the operations in ONFi 2.1 model using full Failures-Divergence refinement checking, rather than just trace refinement. Through the use of compression techniques available in the FDR toolkit and in particular by hiding the events deeper in the model, we were able to get compression of the state-space. The work also reports the number of attempts to compile the full ONFi 2.1 model.

1. INTRODUCTION

The "Grand Challenge in Computing" [10] on Verified Software [23, 11], has a stream focussing on mission-critical filestores, as required, for example, in space-probe missions [14]. Of particular interest are filestores based on the NAND Flash Memory technology, very popular in portable data-storage devices such as MP3 players and datakeys.

This paper follows on from initial formal models of NAND Flash Memory, reported in [3, 1] and then [2] based on the specification published by the "Open NAND Flash Interface (ONFi)" consortium [12]. The first two works looked at the formal model of flash memory in terms of its internal data storage architecture, and the top-level operations that manipulate that storage.

The work in [2] reports on modelling and analysing the finite-state machines in [12] that describe the internal behaviour of flash devices. The modelling was done using machine-readable Communicating Sequential Processes (CSP_M) [21] and the FDR2 tool [8] for the analysis, and was reported in detail in an M.Sc dissertation [4]. The works [4, 2] also describe a methodology for model data-entry based on the "state-chart" dialect of XML (SCXML) using XSLT to translate into CSP. Using XSLT to convert the intermediate XML to CSP saved time and reduced error-proneness in the semi-automatically generated CSP code. The key objective of recent work was to advance the work presented in [4, 2] to cover the full ONFi 2.1 model [13] and to get stronger and more complete results from FDR.

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) – main contribution lies in this section, and conclude (§6).

2. BACKGROUND

A flash memory device is best viewed as a hierarchy of nested arrays of bytes/words, 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. This

*This research was supported by the Programme for Research in Third-Level Institutions (PRTLI4) funded by the Higher Education Authority (HEA),Ireland, through the Lero Graduate School of Software Engineering (LGSSE).

paper focusses on the target level and below, with a particular emphasis on the interactions between LUNs and their containing target.

2.1 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 — in ONFi2.1 there are about 17 of these so we do not list them here. Keeping the size of the model in mind, our CSP model comes in two versions, one covering only the mandatory behaviour, whilst the other also includes the optional operations.

2.2 Host-Target Communication

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

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:

$$CW(readOpcode);\ AW(addr_4);\ \ldots;\ AW(addr_0);\ CW(confirm)$$

The host has then to wait whilst the addressed data is pulled from the relevant page into the selected LUN's page-register, as signalled by the LUN status register. LUN status can be read either directly via an output pin ("hardware" status) or by performing a *ReadStatus* operation ("software" status).

2.3 The ONFi state machines

The internal behaviour of ONFi devices is described by two finite-state machines (FSMs) [13, §7], one describing the behaviour of a target, the other capturing the actions of a LUN. An example state entry, for the target state `T_RPP_ReadParams` (for the *ReadParameterPage* operation) is shown in Fig.1. 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.

3. RELATED WORK

Formal model-checking techniques have been applied to the verification of the Samsung OneNAND flash device driver [17], with particular emphasis on a multi-sector read operation implemented within the FTL. The model-checkers explored were NuSMV, Spin and CBMC. The best tool was reported as CBMC[5], a SAT-solver based model-checker, that works directly with C source code. Follow-on work [18] described the use of a concolic testing method applied to the multi-sector read operation for the flash memory. This method combines a concrete dynamic analysis and a symbolic static analysis to automatically generate test cases and an accordingly exhaustive path testing was performed. Furthermore, the authors compared concolic testing method with other model checking techniques applied to the flash file system domain.

A fully automatic analysis, using Alloy, of a flash filesystem is described in [15, 16]. This was built on top of a simple flash model (at roughly the same level of abstraction as [3]). The basic file operations as well as crucial design features, such as wear leveling and erase-unit reclamation, of NAND flash memory were included in the design. This design also includes a recovery mechanism for unexpected hardware failures. The design was analysed by checking trace inclusion of the flash file system against a POSIX-compliant abstract file system. Similar work, but very much a tools-integration approach to modelling (VDM/HOL/Alloy), was reported in [6, 7]. The key issue here was 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 was used as the main modelling tool, with Alloy and HOL called upon to verify proof obligations that arose.

4. THE CSP MODEL

The main objective of this and previous efforts [4, 2] 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 [8].

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 close correspondence between ONFi states and CSP processes. 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:

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

Modelling the communication between host and target was

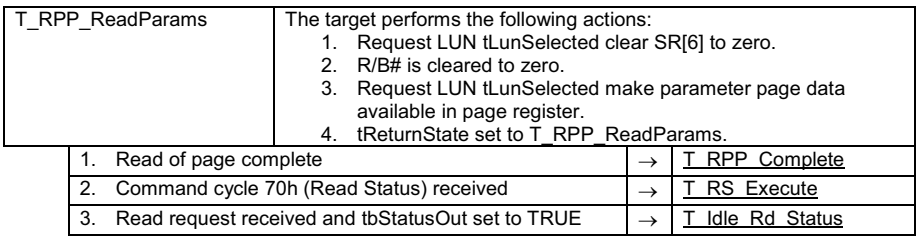

Figure 1: ONFi Target State example [13], Page 175.

straightforward as this is well documented as the external interface of ONFi devices, and had already been modelled in Z at an abstract level[3, 1]. 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).

Certain abstractions and simplifications had to be made so that the FDR2 model-checker could perform analysis without running out of memory. So, 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 8 state variables of the target 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 13 state components. A similar exercise in augmenting the state had to be done for the LUN FSM as well, to a lesser degree (9 ONFi variables were augmented by a further 3).

4.1 CSP Data-Entry

Instead of writing CSP directly, the ONFi finite state machines specifications are described using Statechart XML (SC–XML). This was then automatically converted into CSP via XML Transforms (XSLT) as described in [4, 2]. The model has two versions: the full and mandatory version, covering all the operations and only the mandatory ones respectively. The auto-generated CSP files for the host, target and LUN state-machines vary between the full and mandatory specifications. The other CSP files are hand-crafted and do not vary with the model version. These files are: declarations of datatypes and CSP channels (`header.csp`); status register implementation (`SR6.csp`); internal LUN behaviour (`lun-innards.csp`); and combining all the processes to describe various systems (`footer.csp`). At the top-level, we include all the above files in `ONFI.csp`, (or use `ONFI-mandatory.csp` if only the mandatory model is required).

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. The model validation is described in detail in [2, §5.1]. The verification process undertaken for the FSMs of ONFi 1.0 document is described in [2, §5.2].

Description	ONFi 1.0	ONFi 2.1
Target FSM state variables	7	8
Target FSM state entries	77	88
LUN FSM state variables	8	9
LUN FSM state entries	62	68

Table 1: Comparison of State Variables and Entries for Two Models

5.1 Moving from ONFi 1.0 to ONFi 2.1

First of all, the pre-existing SCXML files of ONFi 1.0 model were updated according to the ONFi 2.1 document [13], after noting the differences between the two versions. This step was straightforward. The comparison of state variables and state entries of the two models are in table 1. This clearly indicates that the ONFi 2.1 model is bigger than the previous version.

5.2 Running the Model in FDR

After conversion, we initially tried to check the model on a Core Duo machine with processor speeds of 2.00GHz and 1.06GHz and 1.75GB of RAM running Ubuntu Linux, and then on a machine having Core 2 Duo Processors of 2.66GHz and 4.00GB of RAM also running Ubuntu Linux. But in each case FDR stopped during its compilation process and halted all the processes running on the CPU. The model ran successfully on a Quad Processor UltraSPARC-IIIi machine with processor speeds of 1.28GHz and 16GB RAM under Solaris. After this experimentation, all the tests were run on this machine. In addition to this, in order to compile the indexed state machines (ISMs) of the model in FDR2, we had to increase the stack size using the command `ulimit -s 262144`. With these settings we were able to compile and verify all required properties on the mandatory-only version of the model. However, all attempts to handle the full model failed, with a compile-time failure. We now describe various attempts made to get the full ONFi model to run through FDR.

5.3 Initial Checks Performed on the Model

The host model in which the status check is done through software was setup as follows:

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

In the case of mandatory ONFi 2.1, when checking the `HOST_SW_TARGET_TWOLUNS` process for deadlock freedom using failures refinement, the comparison of state space of two models reported by FDR is shown in table 2. We see an increase of about 50% in all model-checking size measures reported by the tool.

5.4 More Concrete Tests through Failures Refinement Checks on the Model

In previous work the implementations of Read, Page Parameter, MultiRead and Block Erase operations were tested against their specifications using CSP's Traces model. The implementations of these operations were now tailored so that we could do refinement checks in the more powerful Failures model. As all our models were shown to be divergence-free initially, we did not need to perform full Failures-Divergences refinement checks. For these, the implementation of a process looped back to its specification. For example, the *Read* operation was checked as follows.

In the `READ_SPEC` process we took the `HOST_SW_TARGET_TWOLUNS` process, hid all the events except the host-target read-related commands and data transfers. The timing diagram of these commands and data transfers are specified on Page 127 of [13]. The `POWERON` behaviour is specified as a sequence of first a reset command (FFh) followed by a read status command (70h). The implementation of the Read operation was defined as a process that performed an expected sequences of host target protocol events for a Read (preceded by a `POWERON` behaviour).

```
READ_SPEC = HOST_SW_TARGET_TWOLUNS
       \ diff(Events,union
       ({ht_ioCmd.cmds |
       cmds <-{cmd30h,cmd00h,cmd70h,cmdFFh}}
       ,{|ht_ioDataOut|}))
POWERON = ht_ioCmd.cmdFFh -> ht_ioCmd.cmd70h
       -> ht_ioDataOut.true
       -> SKIP -- poweron events
READ_F_IMPL0 = POWERON;
    ht_ioCmd.cmd00h -> ht_ioCmd.cmd30h
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true
        -- read status returned ready, so read
    -> ht_ioCmd.cmd00h -> ht_ioDataOut.false
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true
    -> READ_SPEC
assert READ_SPEC [F= READ_F_IMPL0
```

The complete list of tests undertaken are in the source file `footer.csp` available on the project website indicated in the acknowledgement section. The specifications and the process implementations have their basis in the ONFi document. All the tests performed on the failures model took 10 to 22 minutes to complete with exception of the first check which took 29 minutes. Some of the important tests performed on the model are listed below:

1. Deadlock and Livelock Freedom Checks using Failures and Failures Divergence Refinement respectively:

```
HOST_SW_TARGET_TWOLUNS :[deadlock free [F] ]
HOST_HW_TARGET_TWOLUNS :[deadlock free [F] ]
HOST_SW_TARGET_TWOLUNS :[livelock free [FD] ]
HOST_SW_TARGET_LUNHIDDEN :[livelock free [FD] ]
HOST_HW_TARGET_TWOLUNS :[livelock free [FD] ]
HOST_SW_ANYCMD_HIDDEN :[livelock free [FD] ]
HOST_SW_ANYCMD :[livelock free [FD] ]
```

2. Correctness Tests for *Read*, *PageParameter*, *BlockErase* and *MultiRead* Operation using Failures Refinement:

```
READ_SPEC [F= READ_F_IMPL0
READ_SPEC [F= READ_F_IMPL1
PP_SPEC [F= PP_F_IMPL0
PP_SPEC [F= PP_F_IMPL1
BE_SPEC [F= BE_F_IMPL0
BE_SPEC [F= BE_F_IMPL1
MULTIREAD_SPEC [F= MULTIREAD_F_IMPL0
MULTIREAD_SPEC [F= MULTIREAD_F_IMPL1
```

3. *Wrong* Implementations to ensure that the tests which should fail must fail:

```
BE_SPEC [F= BE_IMPL_F_WRONG0
BE_SPEC [F= BE_IMPL_F_WRONG1
READ_SPEC [F= READ_IMPL_F_WRONG1
MULTIREAD_SPEC [F= MULTIREAD_IMPL_F_WRONG0
```

5.5 "Deep Hiding" alongwith Model Compression Techniques available in FDR

While dealing with the state space problem, the FDR manual [8] on Page 35 suggests: *"Hide all events at low a level as is possible . . . any event that is to be hidden should be hidden the first time (in building up the process) that it no longer has to be synchronised at a higher level".* The way the model was setup previously, was as follows:

```
LUN(lunID) = diamond(INITIAL_L_IDLE(lunID)
         [| li_events |] LI_IDLE(lunID))
TWOLUNS = LUN(lun0) ||| LUN(lun1)
TARGET = INITIAL_T_POWERON  [| tr_events |]
         READYBUSY(true,true)
TARGET_TWOLUNS = TARGET [| tl_events |] TWOLUNS
```

This clearly shows that the hiding of events was not applied at the first instant of the process setup. We changed the setup of the model as follows:

```
LUN(lunID) = diamond(INITIAL_L_IDLE(lunID)
        [| li_events |] LI_IDLE(lunID)) \ li_events
TWOLUNS = LUN(lun0) ||| LUN(lun1)
TARGET = (INITIAL_T_POWERON  [| tr_events |]
        READYBUSY(true,true)) \ tr_events
TARGET_TWOLUNS = (TARGET [| tl_events |]
        TWOLUNS) \ tl_events
```

Furthermore, by careful investigation we also found that there were two compression techniques i.e. `model_compress` and `normalise` which were neither applied automatically by

Description	ONFi 1.0	ONFi 2.1	Increment Factor
Transitions in ISM	4490300	7023100	156.4%
States Refined	32,338	47,787	147.7%
Transitions during Refinement	78,469	117,473	149.7%

Table 2: Comparison of State Space of Two Models reported by FDR

Description of Model Setting	Time (min)	Nodes	Transitions
High Level Hiding + No Compression Applied	20.6	47787	117473
High Level Hiding + Normalise	18.5	11869	29452
High Level Hiding + ModelCompress	21.5	14696	39455
Low Level Hiding + Normalise	21.76	2393	5292
Low Level Hiding + ModelCompress	23.7	3827	9660

Table 3: Hiding and Compression Techniques Effect on State Space

FDR and nor by us. The remaining compression techniques i.e. `explicate`, `sbsim`, `tau_loop_factor` and `diamond` were already being used in the refinement step of states either by FDR or being manually applied. The details of these compression techniques i.e. how these techniques actually compress the model, are discussed in chapter 5 of [8]. Table 3 lists the impact of these compressions and hiding of events on the state space. Here 'High Level Hiding' refers to the fact that hiding is applied at the top level while 'Low Level Hiding' refers to the hiding being as close to the point of definition of the relevant process as possible. We did these tests using FDR Explorer [9]. These tests were run on a multi-user timeshared machine, but one whose utilisation was pretty low. So, the timings mentioned here are just to indicate that these tests completed in a reasonable time limit.

The use of FDR Explorer was quite straightforward. For example, the commands used for testing `HOST_SW_TARGET_TWOLUNS` were:

```
$FDRHOME/bin/fdr2tix -insecure -nowindow
% source FDRExplorer.tcl
% inspectProcs ONFI-mandatory.csp
  HOST_SW_TARGET_TWOLUNS 0 0
```

After the application of hiding at low level, all the checks were again run to ensure the continued correctness of the model.

5.6 Tackling Full ONFi 2.1 Model

After having confidence that use of FDR Explorer and compression techniques could possibly be helpful in the compilation of the full model, hiding and compression were applied. But it again failed to compile. This is due to the fact that FDR is setup in such a way that it always performs complete ISM generation at the start, and it applies all compressions at a later stage. The failures we encountered occurred in the ISM generation phase. This fact came to our notice when FDR always reported 7023100 transitions (in the case of mandatory ONFI 2.1) in ISM generation in order to perform the first check and after that it started to compress the state-space during each of test runs. So, the application of hiding and compression techniques did not affect the performance.

After this failure, FDR support was contacted to get a 64-bit built of FDR so that we could possibly break the barrier of 32-bit limit for FDR paging. FDR support thankfully provided us with 64-bit built of the tool for solaris machine. But even on 64-bit version of FDR, the ISM generation phase could not complete successfully, giving up after 6 hours of test running whereas the number of transitions at the point of failure was above 23 Million. The status of memory usage was investigated on the solaris machine during the test, just before dying. The machine was consuming more than 30 GB of memory on the local disk as well as 10 GB on the physical memory. This clearly reflects that the full ONFi 2.1 model ISM is too large to handle in present state.

6. CONCLUSIONS & FUTURE WORK

6.1 Conclusions

We are now able to cover many of the operations in ONFi 2.1 model using full Failures-Divergence refinement checking, rather than just trace refinement. For ONFi 1.0, the total count of the CSP code was 1922 of which 1346 were automatically generated from the SCXML sources. Having upgraded to ONFi 2.1, the number of auto-generated lines of CSP has risen to 2070. Through the use of compression techniques available in the FDR toolkit and in particular by hiding the events deeper in the model, we were able to get compression of the state-space, i.e. low level hiding and `normalise` gave approximately 20 times more compression while in case of `model_compress`, this was a factor of about 12. However despite compression tricks and the use of FDR explorer, we still have not been able to compile the full ONFi model, which may represent the current limit of this model-checking technology. This is due to the fact that FDR does full compilation before any compressions are applied.

6.2 Future Work

One possible solution for the full model compilation is to try it on a larger machine with more physical and virtual memory, to handle the high demands made by initial ISM generation.

Another way for dealing with this problem is to analyse the model to see if it can be re-factored into independent chunks as suggested by FDR technical support:

> "Very large individual state machines aren't really FDR's forte: decompose the problem into smaller interacting machines where possible".

The model analysis may require moving away from the SCXML encoding to one that is easier to analyse. The objective of the analysis would be to see if the FSM models can be factored into independent chunks, possibly parametrised, that could be individually checked. More closer liason with the FDR developers might also be helpful.

Yet another unexplored option is converting this model into other related formal languages like CSP|B [22], TCOZ[19], CCS [20]. At the moment the model i.e. SCXML files are far too CSP specific, and would need to be generalised in some fashion.

6.3 Acknowledgments

We'd like to thank Phil Armstrong of Formal Methods (Europe) Ltd., for his assistance with FDR2. This research was supported by the Programme for Research in Third-Level Institutions (PRTLI4) funded by the Higher Education Authority (HEA),Ireland, funded through the Lero Graduate School of Software Engineering (LGSSE).

Sources of the updated model mentioned in this paper are available at the URL: `https://www.cs.tcd.ie/Andrew.Butterfield/Research/FlashMemory`.

7. REFERENCES

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

[2] A. Butterfield and A. Ó Catháin. Concurrent models of flash memory device behaviour. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009Revised Selected Papers*, pages 70–83, 2009.

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

[4] A. O. 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.

[5] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] M. Ferreira, S. Silva, and J. Oliveira. Verifying intel flash file system core specification. In P. L. 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.

[7] M. A. Ferreira and J. N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, pages 153–169, 2009.

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

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

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

[11] T. Hoare, G. T. Leavens, J. Misra, and N. Shankar. The verified software initiative: A manifesto. http://qpq.csl.sri.com/vsr/manifesto.pdf, 2007.

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

[13] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 2.1, ONFI, www.onfi.org, 14th January 2009.

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

[15] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in alloy. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[16] E. Kang and D. Jackson. Designing and analyzing a flash file system with alloy. *International Journal of Software and Informatics (IJSI) 2009*, Vol 3, No. 1, 2009.

[17] M. Kim, Y. Choi, Y. Kim, and H. Kim. Pre-testing flash device driver through model checking techniques. In *ICST*, pages 475–484. IEEE Computer Society, 2008.

[18] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, pages 251–265, 2009.

[19] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, Feb. 2000.

[20] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

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

[22] S. Schneider and H. Treharne. Csp theorems for communicating b machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.

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