

Copyright

by

Art Ó Catháin

2008

# **Modelling Flash Memory Device Behaviour using CSP**

by

**Art Ó Catháin**

A Dissertation submitted to the University of Dublin,  
in partial fulfillment of the requirements for the degree of  
Master of Science in Computer Science

2008

# Modelling Flash Memory Device Behaviour using CSP

Approved by  
Dissertation Committee:

---

---

---

---

---

## Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other university, and that unless otherwise stated, is my own work.

---

Art Ó Catháin

September 5, 2008

## Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

---

Art Ó Catháin

September 5, 2008

# Acknowledgments

I would like to thank my supervisor Dr Andrew Butterfield and second reader Dr Arthur Hughes for their support and encouragement. Philip Armstrong from Formal Systems Europe provided invaluable support for the FDR2 refinement checking tool, without which the project would necessarily have been much more limited in scope.

I would like to attribute the following illustrations used in this document and the presentation poster:

- Yellow lightning bolt by Chris Fry from [openclipart.org](http://openclipart.org)
- Satellite by ivak from [openclipart.org](http://openclipart.org)
- USB stick by Gamer112 from Wikipedia

ART Ó CATHÁIN

*University of Dublin, Trinity College*  
*September 2008*

# Modelling Flash Memory Device Behaviour using CSP

Publication No. \_\_\_\_\_

Art Ó Catháin

University of Dublin, Trinity College, 2008

Supervisor: Dr Andrew Butterfield

Flash memory continues to progress in capacity, speed, and cost, now rivalling magnetic disks in many applications. Lack of standardization prompted leading flash memory manufacturers to form the Open NAND Flash Interface (ONFi) consortium, with the purpose of creating a standard specification.

This project examines the ONFi specification from the perspective of formal methods. Formal verification of the specification will constitute a step towards Grand Challenge 6: Dependable Systems Evolution, which aims to build systems and tools to increase the dependability and reliability of the increasingly ubiquitous computer systems that surround us.

The formal language chosen for the project was Communicating Sequential Processes (CSP), a language designed for analysing concurrency, complete with automatic model-checking tools. The ONFi specification describes a finite state machine for the internals of a compliant device, and it is this FSM that is modelled by the project.

Instead of writing CSP directly, the approach chosen was to convert the FSM's specification into an intermediate form based on State Chart XML, an XML dialect designed for specifying state machines in a machine-readable format. This XML was then automatically converted into CSP via XML Transforms (XSLT). Using XML allowed various other transformations of the specification such as conversion back to HTML to aid checking for implementation errors, and the stripping out of optional parts to leave only the mandatory specification.

Several anomalies and errors were discovered and these were communicated to ONFi. With minor adjustments to correct these, it can be argued that the specification is correct, at least for the mandatory subset of commands.

# Contents

<b>Acknowledgments</b>	<b>vi</b>
<b>Abstract</b>	<b>vii</b>
<b>List of Tables</b>	<b>xii</b>
<b>List of Figures</b>	<b>xiii</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Flash Memory . . . . .	1
1.1.1 NAND and NOR flash . . . . .	1
1.2 ONFi . . . . .	2
1.3 Modelling the ONFi specification . . . . .	2
1.4 Project aims . . . . .	2
<b>Chapter 2 Background</b>	<b>4</b>
2.1 Formal Methods . . . . .	4
2.1.1 Refinement . . . . .	4
2.1.2 Prevalence . . . . .	5
2.2 CSP . . . . .	6
2.2.1 Notation . . . . .	6
2.2.2 Parallel Composition . . . . .	6
2.2.3 Choice . . . . .	7
2.2.4 Communications . . . . .	8
2.2.5 Variables . . . . .	9
2.2.6 Refinement . . . . .	9



2.2.7	Practical implementations . . . . .	11
2.3	Grand Challenges in Computing . . . . .	12
2.3.1	Grand Challenge 6 - Dependable Systems Evolution . . . . .	12
2.4	ONFi consortium . . . . .	14
2.5	ONFi specification . . . . .	15
2.5.1	Physical Interface . . . . .	15
2.5.2	Memory organisation . . . . .	16
2.5.3	Timing . . . . .	17
2.5.4	Commands . . . . .	17
2.5.5	Behavioural flows . . . . .	18
<b>Chapter 3 Design</b>		<b>20</b>
3.1	CSP model . . . . .	20
3.1.1	Host device . . . . .	20
3.1.2	Communications . . . . .	21
3.1.3	Keeping state . . . . .	23
3.1.4	Minimizing state space . . . . .	24
3.2	Analysis . . . . .	24
3.2.1	Deadlock freedom . . . . .	24
3.2.2	Livelock freedom . . . . .	25
3.2.3	Behavioural flows . . . . .	25
<b>Chapter 4 Implementation</b>		<b>27</b>
4.1	Proof of concept . . . . .	27
4.1.1	Host . . . . .	27
4.1.2	Verification . . . . .	27
4.1.3	Limits of prototype model . . . . .	28
4.2	State Chart XML . . . . .	29
4.2.1	XML languages for State Charts . . . . .	29
4.2.2	Origins of SCXML . . . . .	29
4.2.3	W3C specification . . . . .	30
4.2.4	Mapping ONFi specification to SCXML . . . . .	31
4.2.5	Apache Commons implementation . . . . .	33
4.2.6	Adherence to SCXML standard . . . . .	34

4.3	XML Transformations . . . . .	34
4.3.1	SCXML to CSP . . . . .	34
4.3.2	SCXML to HTML . . . . .	35
4.3.3	Mandatory-only states . . . . .	36
4.4	Design of the host process . . . . .	36
4.4.1	Read Command . . . . .	39
4.4.2	Interleaved actions . . . . .	39
4.4.3	Controlling the host . . . . .	40
4.5	Additional processes . . . . .	41
4.5.1	Status Register bit 6 . . . . .	41
4.5.2	LUN ‘innards’ process . . . . .	44
4.6	State Transitions . . . . .	46
4.7	Testing . . . . .	47
4.7.1	Arbitrary commands . . . . .	47
4.7.2	Refinements . . . . .	48
4.7.3	Non-refinements . . . . .	50
<b>Chapter 5 Evaluation</b>		<b>51</b>
5.1	Verification of ONFi specification . . . . .	51
5.1.1	Ready / Busy . . . . .	51
5.1.2	Status Register Update . . . . .	51
5.1.3	Read Parameter Page . . . . .	52
5.1.4	Reset . . . . .	53
5.2	Suitability of CSP for project . . . . .	54
5.2.1	Tractability . . . . .	55
5.3	Success of automation . . . . .	55
5.4	Host process limits . . . . .	55
<b>Chapter 6 Conclusions</b>		<b>57</b>
6.1	Future work . . . . .	57
6.1.1	Optional commands . . . . .	57
6.1.2	ONFi version 2.0 . . . . .	57
6.1.3	Extending and streamlining the SCXML to CSP conversion . . . . .	58
6.1.4	Integration with a verified filesystem . . . . .	58

<b>Bibliography</b>	<b>59</b>
<b>Appendix A Example State Chart XML</b>	<b>61</b>
A.1 Lift . . . . .	61
A.1.1 SCXML representation . . . . .	61
A.1.2 Direct use of Commons SCXML . . . . .	61
A.1.3 CSP representation . . . . .	61
A.2 ONFI . . . . .	65
<b>Appendix B Correspondence with ONFi</b>	<b>66</b>
B.1 SR[6] update . . . . .	66
<b>Appendix C Technical architecture</b>	<b>67</b>
<b>Appendix D Optimizing CSP for FDR2</b>	<b>68</b>
D.1 State space . . . . .	68
D.2 Stack limit . . . . .	69
D.3 Operating system and architecture . . . . .	69
<b>Appendix E Event sequence example: Single ‘Read’ operation</b>	<b>70</b>

# List of Tables

1.1	NAND vs. NOR flash . . . . .	2
2.1	ONFi command set. Mandatory commands are in boldface . . . . .	17
4.1	SR[6] deadlock . . . . .	42
4.2	Sampling SR[6] within target: deadlock remains . . . . .	42
4.3	Expected events for a Block Erase . . . . .	48
4.4	Expected events for a Read . . . . .	49
4.5	Expected events for a Page Program . . . . .	50
5.1	Status Register Update: event sequence for setting SR[6] to 0 . . . . .	51
5.2	Read Parameter Page: deadlock 1 . . . . .	52
5.3	Read Parameter Page: deadlock 2 . . . . .	53
5.4	Reset: deadlock 1 . . . . .	54
5.5	Reset: deadlock 2 . . . . .	54

# List of Figures

1.1	USB memory stick . . . . .	1
2.1	FDR2 refinement checker . . . . .	11
2.2	Sample states from the ONFi specification . . . . .	19
3.1	ONFi specification, p77: State variables for the Target state machine . . . . .	23
4.1	Host Processes for proof of concept model . . . . .	28
4.2	Verifying writes are durable using H_TESTREAD . . . . .	29
4.3	Example State Chart XML . . . . .	30
4.4	Storing a date in State Chart XML . . . . .	31
4.5	Mapping ONFi specification to SCXML: state T_RPP_ReadParams . . . . .	32
4.6	Rendered HTML for three target states. . . . .	37
4.7	XSLT transformations of the Target state machine. Files with dotted lines are generated automatically . . . . .	38
4.8	State diagram for the host process. . . . .	39
4.9	State diagram for the host's Read process. . . . .	40
4.10	State diagram for the host's MultiRead process: testing simultaneous reads on two LUNs	40
4.11	Parallel CSP processes of the entire system . . . . .	43
4.12	Parallel CSP processes of the entire system, modified . . . . .	44
4.13	LUN state L_RD_ArrayRead . . . . .	45
4.14	Target state T_Idle_Read . . . . .	46
4.15	CSP for Read refinement check . . . . .	49
A.1	SCXML for Lift example . . . . .	62
A.2	Using the Apache Commons SCXML library directly with the Lift example . . . . .	63

A.3	Calling the Saxon XSLT processor . . . . .	63
A.4	CSP for Lift example . . . . .	64
A.5	Exploring the CSP process using Probe . . . . .	64
A.6	SCXML vs generated CSP code for state T_RPP_ReadParams . . . . .	65
C.1	ONFI.csp and ONFI-mandatory.csp . . . . .	67

# Chapter 1

## Introduction

### 1.1 Flash Memory

Flash memory devices (for example, USB memory sticks) have become immensely popular in recent years. In embedded devices, flash memory has advantages over other forms of primary storage, including light weight, small physical footprint, reliability, low battery consumption, and lack of moving parts.



Figure 1.1: USB memory stick

#### 1.1.1 NAND and NOR flash

Flash memory is a form of *non-volatile* RAM: it maintains its state even when powered down, unlike conventional computer RAM. There are two types, NAND and NOR, depending on the internal configuration of the memory cells. Which to choose depends on the individual application. Table 1.1 lists the tradeoffs between the two.

NOR's strength is its fine granularity of access: down to the level of individual bytes. NAND flash's minimum granularity is the block level which makes random access slower (the data must be read out serially). NOR tends to find more specialized uses, such as code storage, whereas NAND is popular for general storage purposes.

Inevitably, popularity entails a proliferation of standards and flash memory is no exception.

<b>Metric</b>	<b>NAND</b>	<b>NOR</b>
Cost per bit	Low	High
Capacity	High	Low
Active power consumption	Low	High
Standby power consumption	High	Low
Write speed	High	Low
Read speed	Medium	High
Access	Serial	Random

Table 1.1: NAND vs. NOR flash

## 1.2 ONFi

In a recent effort to bring some standardization to the area, the Open NAND Flash Interface (ONFi) consortium created the ONFi 1.0 Specification. It lays out an interface to which flash memory devices must comply. In future, manufacturers of other devices (MP3 players, mobile phones, and so on) can rely on this interface during the design process, without having to explicitly examine the specification of their chosen brand of flash memory. It is hoped that this will increase interoperability and reduce product design times.

## 1.3 Modelling the ONFi specification

In [1], the authors created a formal model of the ONFi specification, using Z, that corresponds to the external behaviour of an ONFi-compliant device. This model is at a high level, abstracting away much of the detail of the specification. It is, as the authors acknowledge, only a start: the optional ONFi commands are not covered.

## 1.4 Project aims

The specification defines a finite state machine for a device's internal behaviour, which remains to be verified. The project will model this FSM (abstracting certain details, where appropriate) using Communicating Sequential Processes, a formal language for specifying and modelling concurrent programs. The model's behaviour should correspond to the expected external behaviour. This will be verified by checking that certain sequences of events do occur (liveness) and that other sequences are never observed (safety).

The finite state machine's definition is complex: the description takes more than twenty-five pages of text. Therefore another aim of the project is to simplify and if possible, automate, the conversion



of the specification into CSP. If successful, future research could enable arbitrary finite state machines to be analysed in CSP without manual encoding.

If the model is successfully verified, it will contribute to one of the Grand Challenges of Computing: Dependable Systems Evolution. This challenge aims, over a fifteen year period, to develop a suite of tools and reference implementations that allow developers to verify that their programs are correct, using formal methods. One of the targets is a verified file system, which would be implemented on top of a verified storage layer, such as flash memory.

# Chapter 2

## Background

### 2.1 Formal Methods

The term *Formal Methods* encompasses a wide range of methods within computer science, but in general a formal system can be understood as a *specification* for a piece of software or software system, written in a language that allows for automatic reasoning and proof of correctness. Formal languages are grounded in the mathematics of set theory, and thus allow rigorous proofs.

#### 2.1.1 Refinement

Central to formal methods is the notion of *refinement*: incremental implementation of the system from the specification. The original specification should be at a suitable level of abstraction. It is not necessary to include the minute details that will be in the final program.

It is not possible to ‘run’ a formal specification in the way that one might run a piece of code written in a normal programming language. Instead the specification is used to constrain the *implementation*: the actual program code.

It is not feasible, in general, to construct a program directly from its specification, then prove it to be correct. Instead, the program should be constructed in small steps, each time adding more detail. Since the changes are small, it is relatively easy to prove at each stage that the implementation satisfies the specification.

Refinement is used in two ways:

- to prove *safety*: to show that the implementation will only produce the behaviours (or some subset thereof) of the specification.

- to prove *liveness*: to show that the implementation will eventually perform the all the desired behaviours.

Both properties are required for correct implementation. For example, a process that does nothing would trivially satisfy the first property (safety), but would not be much use in real life.

### 2.1.2 Prevalence

The use of formal methods within the software industry is still quite uncommon. Possible reasons include:

**Misconceptions about formal methods** In [2], the author aimed to dispel some of the myths that had grown up around formal methods. The negative myths cited were:

- Formal methods are all about program proving<sup>1</sup>.
- Formal methods are only useful for safety-critical systems
- Formal methods require highly trained mathematicians.
- Formal methods increase the cost of development.
- Formal methods are unacceptable to users.
- Formal methods are not used on real, large-scale software.

**Background of industry practitioners** Many, if not most, people in the software industry do not have a university-level computing degree, and so will not have had the opportunity to learn about formal methods.

**Lack of popular tool support** The advent of Integrated Development Environments (IDEs) such as Eclipse in recent times has made programming an easier task. These IDEs allow the programmer to write, compile, and debug their code all within a single, cohesive environment. Debugging aids, keyword autocompletion, colour syntax highlighting, and so on, have greatly aided the productivity of the modern programmer.

However, despite recent advances [3], an integrated experience is not possible for a formal methods developer, who must still rely on disparate tools.

---

<sup>1</sup>in fact they are all about specification: while proofs of correctness are included, other important parts are specification writing, proving properties about the specification, and constructing programs by manipulating specifications

## 2.2 CSP

Communicating Sequential Processes is a formal language created by Anthony Hoare [4] for the purposes of modelling concurrent, communicating processes. As a formal language, it can be used for creating specifications for software, which can be verified as correct using the CSP model-checking program Failures-Divergence Refinement 2 (FDR2). The software would then be implemented in another more practical language such as Java or C++.

The basic concept in CSP is a *process* which performs *events*. An event is considered to be an atomic, indivisible action. The set of all events that a process performs is known as its *alphabet*.

### 2.2.1 Notation

The convention is to denote events with lowercase letters and processes with uppercase letters, so for example

$$a \rightarrow P$$

is a process that performs the event  $a$ , and then proceeds to behave like the process  $P$ .

CSP can also be written in a machine readable form called  $\text{CSP}_m$ , consisting only of 7-bit ASCII characters. The above process has the following  $\text{CSP}_m$  representation:

$$\mathbf{a \rightarrow P}$$

Processes may be defined recursively, so for example  $P \hat{=} a \rightarrow b \rightarrow P$  is a process that repeatedly performs the sequence of events  $a, b$ .

### 2.2.2 Parallel Composition

CSP's usefulness stems from its ability to put processes in parallel, forcing them to synchronize on certain of their events. A process's *environment* is those events on which it must synchronise with other processes. So the process above, if required to synchronize on the event  $a$ , will wait until its environment is also willing to perform  $a$ , before proceeding like  $P$ . If the environment is unable to perform the event  $a$ , the process is *deadlocked*. In CSP the process named *STOP* is the canonical deadlocked process, unable to perform any events at all. The notation for parallel composition is as follows:

Normal CSP	$\text{CSP}_m$
$P \ A \parallel_B \ Q$	$\mathbf{P \ [ \ A \    \ B \ ] \ Q}$

Here  $P$  and  $Q$  are processes;  $A$  and  $B$  are sets of events on which  $P$  and  $Q$ , respectively, are required to synchronize. The *interface* of a process is the set of events in its alphabet on which it is required to synchronize. If  $A$  is  $P$ 's alphabet and  $B$  is  $Q$ 's alphabet, then the above can be written more simply as  $P \parallel Q$ .

If  $A$  and  $B$  share no common events, then  $P$  and  $Q$  are said to be interleaved. This is written  $P \parallel\parallel Q$ .

### Example

Consider three processes,  $P, Q$ , and  $R$ , as follows:

$$P \hat{=} a \rightarrow b \rightarrow P$$

$$Q \hat{=} b \rightarrow a \rightarrow Q$$

$$R \hat{=} a \rightarrow R$$

$P \parallel R$  will perform  $a, b, a, b, \dots$  and so on.

$P \parallel Q$  will deadlock, since  $P$  is willing to perform only  $a$ , and  $Q$  is willing to perform only  $b$ .

$Q \parallel R$  will not deadlock, since  $Q$  can perform its  $b$  event immediately.  $R$  is not required to synchronize on  $b$  since it is not part of  $R$ 's alphabet.

### 2.2.3 Choice

Any realistic formal language must be able to represent conditional branches in the program flow, and for this CSP uses the notion of choice.

#### External Choice

The following process behaves like either  $P$  or  $Q$ , depending on the next event that takes place in the process's environment:

$$P \square Q$$

If  $P$  and  $Q$  are defined as in the previous section, then the environment's choosing  $a$  will cause the process to behave like  $P$  and also perform  $a$  (then  $b$ , if the environment allows it, since that is  $P$ 's next event).

#### Internal Choice

Processes may also make *internal choice*, in which case the environment has no effect on which path is taken:

$$P \sqcap Q$$

The choice between  $P$  and  $Q$  here is nondeterministic. This non-determinism is often exploited by specifications in order to avoid over-constraining any implementation.

## 2.2.4 Communications

Thus far only events with a single label ( $a, b$ , etc.) have been introduced. CSP allows more sophisticated communication to take place between processes, using *channels*. (In fact,  $a, b$ , above are also channels: the simplest kind of channel, with only one type of event and no data being passed.)

Channels can carry information in the form of integers, booleans, and user-defined types (defined by the `datatype` keyword). They are bidirectional: a process can choose to ‘receive’ or ‘send’ on a particular channel.

### Examples

Channel definition (in  $CSP_M$  notation):

- A channel that passes integers:

```
channel a:Int
```

- A channel that passes both a boolean and an integer value at once:

```
channel b:Bool.Int
```

- A channel that sends a custom datatypes:

```
datatype Languages = english | french | irish  
channel lang:Languages
```

Use of channels by processes:

- A process  $P$ , which repeatedly sends some information on the channel  $b$ , above:

$$P \hat{=} b.false.10 \rightarrow P$$

### Hiding

It is often desirable to ‘hide’ events: define a process with a certain interface, but only expose a subset of that interface to its environment. The remaining events are now *internal*, and are not seen at all by the process’s environment. This is often used during refinement checking (as described in section 2.2.6), to abstract away the irrelevant internal details of an implementation.

## 2.2.5 Variables

Variable declaration and assignment, as used in procedural languages, are not possible in CSP. Instead, state can be maintained by passing parameters to CSP processes.

For example, a simple counter process can keep track of its count as follows:

$$COUNTER(count) \hat{=} up \rightarrow COUNTER(count + 1) \square down \rightarrow COUNTER(count - 1)$$

In  $CSP_M$  this would be written:

```
COUNTER(count) = up -> COUNTER(count+1)
                [] down -> COUNTER(count-1)
```

### Conditional transitions

The guard notation of  $CSP_M$  is a convenient way to express transitions that are conditional on the truth or falsehood of particular expressions. It uses the  $\&$  symbol before the transition. For example, one may wish to extend the counter, above, so that it cannot go below zero. The guard must ensure that *count* is greater than zero before performing a *down* event:

```
COUNTER(count) = up -> COUNTER(count+1)
                [] (count > 0) & down -> COUNTER(count-1)
```

## 2.2.6 Refinement

CSP supports refinements (as described in section 2.1.1) and the model-checker FDR2 can automatically prove (or disprove) that one CSP process is a refinement of another.

### Digression: Traces, Failures, and Divergences

The simplest way to characterize a CSP process is to enumerate its possible behaviours. A *trace* of a process is a sequence of events that the process is capable of performing. This includes the null sequence, since at the time of observation, the process might not yet have performed any events. Two processes can therefore be compared by comparing their respective sets of traces.

This characterization is not sufficiently powerful however, as the following example demonstrates. Consider the two processes *P* and *Q*:

$$P \hat{=} \left( \begin{array}{l} a \rightarrow b \rightarrow STOP \\ \square \\ a \rightarrow c \rightarrow STOP \end{array} \right) \quad Q \hat{=} a \rightarrow \left( \begin{array}{l} b \rightarrow STOP \\ \square \\ c \rightarrow STOP \end{array} \right)$$

Are  $P$  and  $Q$  equivalent? Their sets of traces are identical, but they will not interact identically with an environment that first performs the event  $a$ . At the first  $a$  event,  $P$ 's future evolution is determined by which path is taken. Since both paths start with an  $a$  event, either can be taken and the choice is non-deterministic, i.e. made by  $P$  internally. If the top process is chosen,  $P$  can only then perform  $b$ , and if the bottom process is chosen,  $P$  can only then perform  $c$ . This is in contrast to  $Q$ , which remains capable of performing either  $b$  or  $c$  after the initial  $a$ .

Thus we introduce the notion of *refusals*: the set of events which a process will refuse at a given point in its execution. A process's *failure* is a pair, containing a trace of events, and its refusals at that point in the trace. The set of all failures can characterize a process more fully than just its traces.

There remains one final consideration. A process may continually perform internal events, without interacting with its environment. This is subtly different from deadlock, and is known as *livelock*. A livelocked process will have trace with an infinite sequence of internal events. Such a trace is known as a *divergence*.

In increasing order of power, then, are the three models of CSP:

Model	Characterization
Traces	Traces
Stable Failures	Traces, Failures
Failures-Divergence	Traces, Failures, Divergences

To say a process is a refinement of a specification means that its characterization set is a subset of the specification's. Proving this can be done by hand, but generally computer assistance is required for complex implementations.

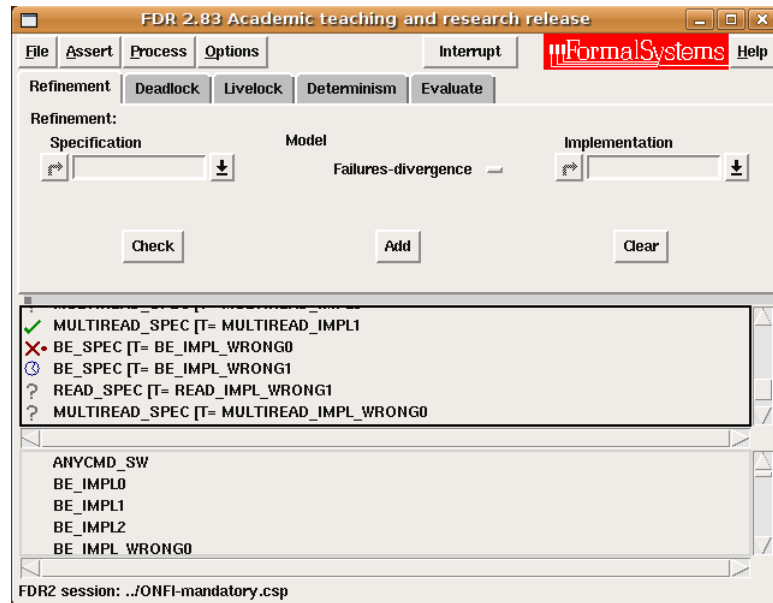
## FDR2

This application performs refinement checks for all three CSP models. CSP models must be encoded in  $CSP_M$ : an appendix to the FDR2 manual provides a comprehensive description of the language's syntax. FDR2 runs on Linux and Solaris. Figure 2.1 shows a screenshot.

A naive implementation of a refinement checker would fully expand the state space of the two processes, then check each pair of states. Early versions of FDR did this, but the program has been enhanced to allow compression of the state-space, mitigating the tendency towards exponential expansion. It also now supports 'lazy' exploration of the state space, only expanding those paths necessary.

It will also check for deadlock in the Failures model and livelock in the Failures-Divergence model. These checks are in fact themselves refinements:





The green tick indicates a successful refinement; the red cross indicates the opposite. The clockface indicates a refinement check in progress, while the question marks are refinement checks that have not yet been set in motion by the user.

Figure 2.1: FDR2 refinement checker

- a deadlock-free process will refine  $DF$ , the most non-deterministic non-deadlocking process, defined as

$$DF \hat{=} \prod_{i \in Events} i \rightarrow DF$$

where  $Events$  is the set of all possible events.

- a livelock-free process will refine  $CHAOS$ , a special process that can perform any event or refuse any event, but never diverges.

FDR2 can also check whether a system is *deterministic*, useful for security analysis.

### 2.2.7 Practical implementations

One of the difficulties in using formal methods is ensuring not only that the implementation is a correct refinement of the specification, but that the implementation itself is correctly described using the formal language. This difficulty is compounded by semantic differences between modern programming languages and formal languages.

A way around this problem is to use a language such as J-CSP [5]. This set of Java classes

allows a fully functioning Java program to be created, but within the CSP framework. As long as all inter-process communication takes place using the special Channel classes, the program's possible behaviours are guaranteed to mirror those of the CSP process on which it is based. The programmer does not have to deal with the complexity of Java's wait / notify multithreading paradigm, and the possibility of unnoticed errors is much reduced.

## 2.3 Grand Challenges in Computing

A Grand Challenge, according to the UK Computing Research Committee [6], is

a goal that is recognized one or two decades in advance; its achievement is a major milestone in the advance of knowledge or technology, celebrated not only by the researchers themselves but by the wider scientific community and the general public.

In 2002 this committee was formed and called for ideas for a set of such challenges, to be discussed and then formalized. In early 2003, it published seven challenges, of which number six, *Dependable Systems Evolution*, is a key motivator of this project.

### 2.3.1 Grand Challenge 6 - Dependable Systems Evolution

With the increasing ubiquity of computing in all aspects of life comes increased reliance on defect-free software. Arguably, advances in the reliability and dependability of software have lagged other, more noticeable, advances such as the relatively new object-oriented programming and software-as-a-service paradigms. Nor have the substantial increases in hardware reliability been mirrored in software.

#### Applications

**Safety critical systems** are the obvious application of this challenge, and research has been in progress for many years [7], in such areas as aviation, railway systems, nuclear power, and medical systems.

**Embedded systems** are another area requiring high software dependability. Many of these systems are deployed in places that makes applying patches to fix bugs difficult and expensive. For example, the software on NASA's space probes must function reliably for many months, and failure may lead to loss of critical scientific data or even the entire mission.

**The telecommunications backbone** requires reliability of the order of  $10^{-9}$ . Though fault-tolerance techniques such as redundancy are the primary method of achieving this, eliminating software defects will also contribute to this goal.

**Electronic commerce** is now a significant sector of the economy. Though not safety-critical, failure of a web server for even a short period can lead to large losses to a company as customers, unsuccessful on its site, make their purchases on that of a competitor.

Current approaches to producing software free from defects vary. The ad-hoc, build-and-fix approach is still prevalent, particularly among programmers without much formal training. Software companies tend to be more professional and employ techniques such as unit testing and acceptance testing. These testing techniques are best employed throughout the entire software lifecycle, if possible.

While testing has, and will continue to have, an important role in reducing software errors, it suffers from being somewhat unscientific. Its success depends on the appropriate tests being created in the first place. Only an approach based on formal methods can provide rigorous proof of the correctness of a given piece of code.

Software is increasingly used not on its own, but as part of a larger ecosystem of interacting software components. This is evident on the internet, for example in Web Services. This complexity makes extensive testing more difficult. Some systems are created on-the-fly, and may only have a short expected lifespan. Constructing these systems will be much easier if each component has been verified and can exhibit guaranteed behaviour.

## Tools

This vision of dependable systems is at the heart of Grand Challenge 6, and the challenge, in [8], describes the tools that will be necessary for its realization.

Today's software specification is usually done using long form documents, designed to be read by humans rather than computers. This specification is converted by hand into code. *Model-driven software development* aims to automate much of this process, by constraining specifications into a machine-readable framework. This allows automatic conversion into code or code templates. Non-functional requirements (e.g. timing, resource usage) can also be included. The technology has already moved from theory into practical deployment.

One of the key tools is the *verifying compiler*: a compiler that integrates code generation with verification of the correctness of the resulting code. This is achieved using assertions at key interfaces, inserted by the programmer during the coding stage.

The compiler can be supplemented by an *invariant generator*. This allows specifications to be automatically extracted from pre-existing code, by discerning program invariants.

*Testing* will continue to play an important role, but will be enhanced by automatic generation of test cases. Both functional and non-functional requirements will be tested.

Model-checking and theorem-proving tools will aid *automated verification of software properties*. Many of these tools will build on existing formal methods tools; it is possible that FDR2, as used in this project, will play its part in future, albeit in a different and possibly unrecognizable form.

### GC6 and ONFi

All large scale software systems require a file system. In [9] the authors constructed and verified a simplistic Unix-like file system, demonstrating the feasibility of formal methods in this context. Such verification is clearly worthwhile: in [10] the authors applied model-checking to three common file systems (ext3, JFS, ReiserFS) and found bugs in all of them. These bugs were non-trivial, and the maintainers considered them serious enough to release patches within days.

However to the author's knowledge, no similar task has been undertaken for the hardware part of file storage. NASA's Laboratory for Reliable Software have proposed, in [11], developing a NAND-flash based verified file system, under the umbrella of Grand Challenge 6. The ONFi specification is an ideal candidate for this: its state machine notation, described in the next section, makes it relatively easy to check with formal methods.

## 2.4 ONFi consortium

The Open NAND Flash Interface consortium [12] exists to

develop a standardized NAND Flash interface that allows interoperability between NAND devices

Founded in May 2006, it comprises mainly semiconductor and other hardware industry representatives. The principal members are Hynix, Intel, Micron, Phison, Sony, and ST Microelectronics, though Samsung, the largest manufacturer of flash memory [13] is not a member.

The specification's aims included

- forwards compatibility: new innovations can be accommodated within the existing framework.
- backwards compatibility (partial): the design is consistent with existing NAND flash designs, to allow ease of transition.

- interoperability: device behaviour will no longer be manufacturer-dependent.
- parameterization: devices will describe their features to the host using parameter pages, avoiding the requirement for hard-coding chip ID tables within their hosts.

The first specification (ONFi version 1.0) was released in December 2006. It was partly based on behaviours from existing NAND flash devices from Hynix, Micron, and ST.

The followup version 2.0 was released in February 2008. Enhancements are primarily related to speed: the spec now allows for up to 133MB/second, increased from 50MB/second in ONFi v1.0.

## 2.5 ONFi specification

The specification document is intended to provide all information required to produce an ONFi-compliant flash memory chip. It is divided into discrete sections, covering the physical interface, memory addressing, timing, commands, and behavioural flows. These are summarized below.

**Note:** To avoid confusion, sections within this document are referred to using normal text, e.g. section 4.5. When referring to sections within the ONFi specification, a sans-serif font will be used, e.g. section 7.

### 2.5.1 Physical Interface

ONFi devices can be produced in several different chip packages. The specification defines the pinout for these different package types. The pinout varies depending on whether the device supports 8 or 16 bit data transfer. Also specified here is the meaning of the various signal pins, and maximum voltage ratings.

The most important signals for this project are as follows:

**Ready / Busy** This signals to the host whether the device is currently busy with one or more flash data operations. This signal is used by a hardware host. A software-based host must use a special command, Read Status, instead.

**IO0 - IO7** These pins are used for input and output of 8-bit wide bytes. There are also IO8 - IO15 for devices supporting 16-bit data transfer, but this project does not consider that level of detail.

**Read Enable** Since the I/O pins are bidirectional, this pin signals that the host is performing a read; i.e. the device should output data.

**Write Enable** The reverse of Read Enable; the device should prepare to receive data on the I/O pins.

**Address Latch Enable** This signals that the incoming data on the I/O pins is an address byte.

**Command Latch Enable** This signals an incoming command byte.

**Write Protect** As might be expected, this disables Page Program and Erase operations on the device.

The rest of the detail of this section is of only tangential relevance to the project.

## 2.5.2 Memory organisation

ONFi devices comprise one or more *targets*. Targets are themselves divided into one or more *logical units* (LUNs). The LUNs are capable of executing commands independently of each other. With appropriate interleaving of data, this allows greater throughput, as well as simultaneous reads and writes to different memory locations.

All host communication takes place at the target level: there is no direct access to the LUNs, though the host device must be aware of the division of the flash array into separate LUNs.

### Page Register

The flash array is not accessed directly by a host. Instead, the *page register* acts as a buffer. The page register consists of fast non-persistent RAM. To read data, the host requests the LUN to copy the data from the flash array to the page register. This is a relatively slow operation, so the host must wait until the LUN signals the operation is complete before attempting to read from the page register.

Similarly, to write data, the host writes the data to the page register, then the LUN copies it to the appropriate location in the flash array.

### Blocks and Pages

Within the LUN the memory is first divided into blocks and then into pages. The page, comprising a number of bytes or words, is the smallest addressable unit for read and write operations. Due to the physical nature of flash memory, erase operations can take place only at the block level. The specification places restrictions on the number of blocks and the number of pages per block, due to the nature of binary addressing.

<b>Read</b> , Copyback Read, <b>Change Read Column</b>
Read Cache Enhanced, Read Cache, Read Cache End
<b>Block Erase</b> , Block Erase Interleaved
<b>Read Status</b> , Read Status Enhanced
<b>Page Program</b> , Page Program Interleaved, <b>Change Write Column</b>
Copyback Program, Copyback Program Interleaved
<b>Read ID</b> , <b>Read Parameter Page</b> , Read Unique ID, Get Features, Set Features
<b>Reset</b>

Table 2.1: ONFi command set. Mandatory commands are in boldface

## Addressing

The address of a given byte or word within a target is thus a combination of:

- LUN address
- block address
- page address
- location within the page

Each of these address is described with one or more octets. The combination of (block + page) address is known as the *row address*, and the location within the page is known as the *column address*.

### 2.5.3 Timing

ONFi devices must support certain minimum and maximum timing parameters. There are several different timing modes (sets of these parameters), from mode 0 (slowest) to mode 5 (fastest). Only mode 0 is mandatory. This flexibility allows the specification to support both low and high performance devices.

Details of timing are not relevant to this particular project since asynchronous CSP has no concept of time.

### 2.5.4 Commands

The commands available are divided into mandatory and optional sets, shown in Table 2.1. The parameter page is used by the host to ascertain which of the optional commands are supported by a given ONFI-compliant chip.

Commands must be entered via a certain sequence of signal pin activations and I/O operations; these sequences are illustrated in graphical form.

## Status Register

Each LUN maintains a status register byte. The value of certain of its bits indicate:

Bit	Value
0	failure of the last command
1	failure of the next-to-last command
5	whether an array operation is in progress
6	whether the LUN is ready to receive another command. This bit (Status Register 6) is ANDed together with the SR[6] <sup>1</sup> bits from the other LUNs and the result is signalled on the target's Ready/Busy pin.
7	whether write-protected

Other bits are reserved.

### 2.5.5 Behavioural flows

This section contains a detailed description of finite state machines for both target and LUNs. It is not in a machine readable format, requiring translation to an intermediate format before it can be analysed by computer.

Figure 2.2 is an example of the format of each state, in this case from the Target state machine. The first line contains the name of the state, and any actions performed on entry to that state. Remaining [numbered] lines describe, in order of priority, possible transitions from that state. In this example, tCopyBack, tReturnState, and tbStatus78hReq are state variables, stored within the target.

---

<sup>1</sup>here we follow the convention of the specification document and write SR[6]. This is not a citation mark.



T_Idle	tCopyback set to FALSE. tReturnState set to T_Idle.	
1. WP# signal transitioned	→	<a href="#">T_Idle_WP_Transition</a>
2. LUN indicates its SR[6] value transitioned	→	<a href="#">T_Idle_RB_Transition</a>
3. Command cycle received	→	<a href="#">T_Cmd_Decode</a>

T_Cmd_Decode <sup>1</sup>	Decode command received. If R/B# is set to one and command received is not 70h (Read Status), then tbStatus78hReq is set to FALSE.	
1. (Command 80h (Page Program) or command 60h (Block Erase) decoded) and WP# is low	→	<a href="#">T_Idle</a>
2. Command FFh (Reset) decoded	→	<a href="#">T_RST_Execute</a>
3. Command 90h (Read ID) decoded	→	<a href="#">T RID_Execute</a>
4. Command ECh (Read Parameter Page) decoded	→	<a href="#">T_RPP_Execute</a>
5. Command EDh (Read Unique ID) decoded	→	<a href="#">T_RU_Execute</a>
6. Command 80h (Page Program) decoded and WP# is high	→	<a href="#">T_PP_Execute</a>
7. Command 60h (Block Erase) decoded and WP# is high	→	<a href="#">T_BE_Execute</a>
8. Command 00h (Read) decoded	→	<a href="#">T_RD_Execute</a>
9. Command EFh (Set Features) decoded	→	<a href="#">T_SF_Execute</a>
10. Command EEh (Get Features) decoded	→	<a href="#">T_GF_Execute</a>
11. Command 70h (Read Status) decoded	→	<a href="#">T_RS_Execute</a>
12. Command 78h (Read Status Enhanced) decoded	→	<a href="#">T_RSE_Execute</a>
NOTE:		
1. The host shall ensure R/B# is set to one before issuing Target level commands (Reset, Read ID, Read Parameter Page, Read Unique ID, Set Features, Get Features).		

Figure 2.2: Sample states from the ONFi specification

# Chapter 3

## Design

The project has two identifiable stages: creation of a CSP model, and its subsequent analysis. There is, of course, a degree of overlap between these stages. Choices made during the design of the CSP model are informed to some extent by the results of analysis.

Any problems identified during analysis should be classified as either:

- errors in the CSP model, and hence corrected, or
- potential inconsistencies within the ONFi specification itself. These require further discussion with ONFi.

### 3.1 CSP model

The state machine notation of the ONFi specification allows for relatively direct conversion into CSP: there is a one-to-one mapping 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 on each other's. In CSP notation this is written (for a single target and two LUNs):

$$\begin{aligned} LUNS &\cong LUN(0) \parallel \parallel LUN(1) \\ SYSTEM &\cong TARGET \parallel LUNS \end{aligned}$$

#### 3.1.1 Host device

There are various restrictions on the allowed behaviour of a host that interacts with an ONFi target: for example, it is not always permissible to issue a Read Status command. Therefore it is necessary

to introduce a third process to mimic a host that is acting in accordance with those restrictions:

*HOST* || *TARGET* || *LUNS*

Additionally, during the analysis phase, it will be used to verify, for example, that page program and erase operations are durable, in other words that bytes that are programmed stay programmed.

### **Hardware vs. software host**

The specification envisions two separate ways of communicating with a target: hardware or software. A hardware host will monitor the Ready/Busy pin of the package while waiting for flash commands to complete. For a software host, which cannot perform such monitoring, two special commands are provided: *Read Status*, and *Read Status Enhanced*.

**Read Status**, when issued to the target, outputs a byte which corresponds to the Status Register of the last LUN on which a command was issued.

**Read Status Enhanced** is a more sophisticated command which allows a host to query the ready status of an arbitrary LUN. This is useful when interleaved operations are taking place on multiple LUNs. A hardware host must also use this command when querying an arbitrary LUN: the ready / busy signal would not be sufficient.

### **3.1.2 Communications**

Note: the following convention is used in this project for CSP channels:

*ht\_XXXX*: host–target interactions

*t1\_XXXX*: target–LUN interactions

#### **Host to target**

Appropriate CSP events to model the communications between the host device and the target can be gleaned from the descriptions of each state.

For example, consider the states in Figure 2.2. *T\_Idle*'s first transition, “WP# signal transitioned” is caused by a change to the WP# pin of the target. It would be triggered by the host because of some user action. Therefore it corresponds to a CSP channel between the host and the target, containing a boolean (for Write Protect *on* or *off*).

## I/O channel

The I/O pins are always used with some combination of the Address Latch Enable, Command Latch Enable, Read Enable, and Write Enable pins. They are modelled, therefore, as separate channels for each of these different types of event:

**ht\_ioCmd** - host command input

**ht\_ioAddress** - host address input

**ht\_ioDataIn** - host data input

**ht\_ioDataOut** - target data output

Since host and target are forced to synchronise on these channels, deadlock will occur if, for example, the host attempts to input an address byte when the target is expecting a data byte.

## Read channel

Some confusion may arise between the read *command*, and the read *signal*.

**Read Command (00h)** is input to the target along the I/O pins. It instructs the LUN to begin copying data from a specified location in memory to the page register.

**Read Signal** is a pin that is triggered when the host desires to read data, either from a LUN's page register or from its Status Register byte. It is triggered at some point *after* a read command has been issued.

It is this latter that is modelled as a channel, **ht\_read**.

## Target to LUN

Somewhat more ambiguous are the communications between the target and the LUNs - probably because the authors of the specification wish to leave much of this implementation detail to the manufacturers.

For example, during a Page Program 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".

It is assumed that the target can transfer the address in one operation, rather than byte-wise. The communications channel for the above command is thus defined as

<b>tbStatusOut</b>	This variable is set to TRUE when toggling of RE# should return the status value. The power-on value for this variable is FALSE.
<b>tbChgCol</b>	This variable is set to TRUE when changing the column is allowed. The power-on value for this variable is FALSE.
<b>tCopyback</b>	This variable is set to TRUE if the Target is issuing a copyback command. The power-on value for this variable is FALSE.
<b>tLunSelected</b>	This variable contains the LUN that is currently selected by the host. The power-on value for this variable is 0.
<b>tLastCmd</b>	This variable contains the first cycle of the last command (other than 70h/78h) received by the Target.
<b>tReturnState</b>	This variable contains the state to return to after status operations.
<b>tbStatus78hReq</b>	This variable is set to TRUE when the next status operation shall be a 78h command (and not a 70h command). The power-on value for this variable is FALSE.

Figure 3.1: ONFi specification, p77: State variables for the Target state machine

```
channel t1_programRequest : Lun.Bool.Bool.Bool.Bool
```

where each Bool is an address ‘byte’ (modelled as a bit, see section 3.1.4 below).

There are similar channels for the Erase and Read commands. Other commands that do not have an associated address are passed on a generically defined

```
channel t1 : Lun.LunEvent
```

where LunEvent is a user-defined set of all the remaining commands.

### 3.1.3 Keeping state

Various state variables are defined in the specification, at the start of the state machine descriptions (section 7). Figure 3.1 shows the variables of the Target state machine. There is a similar definition for the LUN. Each of these variables is later referred to at one point or another. A desired outcome of the project is to ensure that there is no ambiguity in the specification, no implicit state required that should be made explicit.

For example, the host can trigger the Write Protect pin to signal to the target to ignore any Page Program or Erase events. The target checks the Write Protect pin before either of these operations, and ignores them if appropriate. In CSP we must store the state of the write protect pin each time it changes. The same principle applies to the Ready/Busy pin.

Another implicit piece of state that must be stored in addition to those already defined is the value of each byte of data in the flash array. For a realistic flash memory device this would be very large. The next section outlines the steps taken to minimize the amount of state required.

### 3.1.4 Minimizing state space

Certain abstractions and simplifications have to be made so that the FDR2 model-checker can perform analysis without running out of memory.

Rather than model a full 8 bits (with a possible  $2^8 = 256$  different events), the I/O channel is restricted to the set of known command types.

#### Addressing

In section 2.5.2, the address of each data byte stored within the device was presented as the combination of LUN, block, page, and column address. The host must input the address to the target in a piecewise manner, with each of the components being a multiple of eight bits in size.

To model the specification in its full generality would imply a CSP model with around  $2^{8 \times 4} = 4$  gigabits of state, which is clearly computationally intractable. Instead each of these components is simply modelled as one bit. The column address uses two bits so that the ‘Change Read Column’ command, which expects more than one address byte to be input, is modelled correctly.

This results in 5 address bits (1 LUN, 1 block, 1 page, 2 column), which can address  $2^5 = 32$  locations of data. The flash array is abstracted such that each location stores only a single bit rather than a byte. This is at least theoretically tractable.

The decision to use bits instead of bytes means that the I/O channels mentioned in section 3.1.2 each carry a single bit. For example the  $CSP_M$  declaration for the ‘host data input’ channel will be:

```
channel ht_ioDataIn : Bool
```

#### Mandatory commands

As previously discussed, certain parts of the command set are optional. Initial efforts focus on modelling the mandatory commands only, but within a framework that allows the model to be easily extended to the full specification.

## 3.2 Analysis

### 3.2.1 Deadlock freedom

The Host + Target + LUNs combination of processes should be deadlock free. The Target + LUNs combination is expected contain deadlocks, since the specification forbids certain sequences of actions (e.g. a Read Status command when `tbStatus78hReq` is set to TRUE).

### 3.2.2 Livelock freedom

The Host + Target + LUNs combination of processes should be livelock-free.

When the target to LUN events are hidden, such that the only visible events are between the host and the target, the model should still be livelock free. If this were not the case, it would imply that there are sequences of events in which the host waits indefinitely for a response from the target, which is undesirable.

### 3.2.3 Behavioural flows

Deadlock and livelock freedom are not enough, in general, to prove the system works as expected. It is necessary also to prove that certain behaviours are possible, and that others are forbidden.

#### Why deadlock and livelock freedom are not enough

Consider the process  $P \hat{=} a \rightarrow P \square b \rightarrow P$ , which is in parallel with a larger, more complex system  $Q$ , whose behaviour is too complex to write down in a human-comprehensible form, and includes many hidden events. One may wish to prove that  $P$ , in this environment, sometimes performs both *a* and *b*.

$P \parallel Q$  may well be deadlock free. This tells us only that the system never reaches a state which is incapable of performing *any* event. However,  $Q$  could, for example, perform behaviour similar to the process  $Q \hat{=} a \rightarrow Q$ . Then an observer of the system would see a continual sequence of *a* events, without ever seeing a *b* event. Worse still,  $Q$  might simply livelock:  $P$  would never perform any events.

#### Approach

These limitations can be surmounted in a number of ways.

**Refinement** Create a CSP process that shows the desired behaviour, and then prove that it is a refinement of the model. It is not necessary for the new process to include every event: one can hide the inner details of the model before presenting it as a specification to be refined.

An incorrect refinement should fail. This can be exploited by deliberately creating a refinement that displays behaviour that should not be possible. FDR2's check should fail, providing a counter-example which can be inspected to confirm that the failure is in the expected place.

**Parallelism** Create a CSP process that chooses, nondeterministically, possible behaviours. This process is placed in parallel with the model. On hiding all other events, the model should be

both deadlock and livelock free. This implies that whatever behaviour is chosen, the model can always exhibit that behaviour.



# Chapter 4

## Implementation

### 4.1 Proof of concept

The first proof-of-concept model was hand-written in CSP, for a reduced specification with only

- basic command flows corresponding to Page Program and Read,
- a single target and a single LUN, and
- two address bits, giving four bits of flash storage.

This model was developed at the same time as a spreadsheet giving a human-readable presentation of the sequence of events on both target and LUN. A sample of the spreadsheet is presented in Appendix E.

#### 4.1.1 Host

The Target and LUN are placed in parallel with a Host process, as described in section 3.1.1. The host is modelled by a number of processes, each of which is called at the appropriate time. These processes are described in Figure 4.1.

#### 4.1.2 Verification

##### Deadlock and livelock

The simplistic model was verified to be deadlock and livelock free. Since it contained only a subset of the available event sequences, this result was neither surprising nor particularly valuable.

- H\_POWERON : Performs the required reset command on initializing an ONFi device.
- H\_PAGEPROGRAM : Taking three parameters (two address bits and a data value), inputs the correct sequence of commands to the target in order to program the requested address with the given data value.
- H\_READ : Taking two parameters (address bits), inputs the correct sequence of commands to the target in order to read data from the requested address.
- H\_TESTREAD : Performs the same sequence as H\_READ, but takes an additional parameter: the expected value of the data at the chosen address. This process will deadlock if the data returned is different from that expected.
- H\_ANYCOMMANDS : Repeatedly chooses to perform either a read or a page program at the address (0,0). Each time, the decision is nondeterministic (i.e. internal choice).

Figure 4.1: Host Processes for proof of concept model

### Other properties

It was possible to prove that writes to the flash array were durable using H\_TESTREAD, using processes illustrated in Figure 4.2. The process H\_TEST\_PP\_THEN\_READ, when put in parallel with the Target and LUN process, is deadlock free.

To check that H\_TESTREAD definitely does deadlock when an unexpected data value is returned, H\_TEST\_PP\_THEN\_READ\_SHOULD\_DEADLOCK writes 1 to a location and then checks that H\_TESTREAD deadlocks when given 0 as the expected value.

### 4.1.3 Limits of prototype model

CSP written by hand tends to be laborious and error prone for a substantially sized project. Adding a state variable necessitates changes to every single CSP process and every transition within that process, for example.

The bulk of the project consisted of converting the specification's state machine (as described in section 2.5.5) into CSP. To assist the automation of this process, the specification was first converted by hand to an intermediate XML form. This XML was then converted, using XSLT, into CSP.

```

H_TEST_PP_THEN_READ =
    -- after poweron, all data = 0
    H_TESTREAD(data0,addr0,addr0);
    H_TESTREAD(data0,addr0,addr1);
    H_TESTREAD(data0,addr1,addr0);
    H_TESTREAD(data0,addr1,addr1);
    H_PAGEPROGRAM(data1,addr1,addr0); -- write 1 to address 10
    H_TESTREAD(data0,addr0,addr0);
    H_TESTREAD(data0,addr0,addr1);
    H_TESTREAD(data1,addr1,addr0); -- check write correctly changed data0 to data1
    H_TESTREAD(data0,addr1,addr1);
    H_PAGEPROGRAM(data0,addr1,addr0); -- write 0 to address 10
    H_PAGEPROGRAM(data1,addr0,addr1); -- write 1 to address 01
    H_TESTREAD(data0,addr0,addr0);
    H_TESTREAD(data1,addr0,addr1);
    H_TESTREAD(data0,addr1,addr0);
    H_TESTREAD(data0,addr1,addr1)

H_TEST_PP_THEN_READ_SHOULD_DEADLOCK =
    H_PAGEPROGRAM(data1,addr1,addr0);
    H_TESTREAD(data0,addr1,addr0) -- data1 is returned here, not data0

```

Figure 4.2: Verifying writes are durable using H\_TESTREAD

## 4.2 State Chart XML

### 4.2.1 XML languages for State Charts

The Unified Modelling Language (UML), a visual language for conveying system descriptions, defines various types of diagrams, including one for state machines. Since UML may be stored in an XML format called XSI (XML Schema for Interchange), this provides another way of writing state charts in XML. However XSI, while theoretically human-readable, tends to be very verbose.

State Chart XML (SCXML) is the alternative chosen for the project: a reasonably compact (considering XML's general verbosity) and easily decipherable format for describing state machines.

### 4.2.2 Origins of SCXML

State Chart XML evolved from Call Control XML, a language designed to assist and standardize the handling of voice calls during, for example, audioconferencing. It also incorporates elements of Harel State Tables [14]. Despite its specialized roots, SCXML is a fully general way of describing finite state machines. It has reached the status of W3C Working Draft [15].

```

<state id="thisState">
  <onentry>
    <!-- events can be raised and assignments to data items can take place here -->
  </onentry>
  <transition event="some_event" target="nextState">
    <!-- this transition takes place if some_event occurs. Assignments and
    events can take place here, that are unique to this transition -->
  </transition>
  <onexit>
    <!-- again, events and assignments can be placed here. These occur
    regardless of the particular transition taken -->
  </onexit>
</state>

```

Figure 4.3: Example State Chart XML

### 4.2.3 W3C specification

The specification defines the XML tags found in a valid SCXML document, and their semantics. It is modular in nature, divided into the core module, and others for external communications, data storage, and scripting.

It also provides a reference algorithm for an implementation and some example SCXML documents.

#### Elements

The core module defines the valid XML elements and attributes within an SCXML document. The most important of these is the `<state>` element, of which an example is presented in Figure 4.3. The tag names should be self-explanatory.

The system transitions from state to state, depending on the events that trigger each particular transition. Events can be raised both before and after a state is active using the `<onentry>` and `<onexit>` elements. Conditional expressions can be used in both transitions and entry/exit procedures using the `<if>` element. Parallel states are possible, and events raised by one state can trigger events in another.

There are various enhancements such as substates, communication with external processes, and a history function, but consideration of these is not necessary for this project.

#### Data storage

Data can be stored persistently between states. As might be expected in an XML language, data is stored in a tree structure. The format of the tree is not constrained, other than to require a top-level

```

<datamodel>
  <data name="date">
    <year/>
    <month/>
    <day/>
  </data>
</datamodel>

```

Figure 4.4: Storing a date in State Chart XML

<datamodel> and below that, <data> elements. This affords the system designer both simplicity and flexibility.

Figure 4.4 gives an example for storing a date. It is defined at the start of the document under a single <datamodel> tag. There is no scoping of variables: any state may modify any data item, though for the creator's convenience, the data declaration may in fact be distributed throughout the document.

Assignment is simply a case of using the <assign> element, for example:

```
<assign location="year" expr="2008"/>
```

Variables may be queried before a transition using the <cond> attribute, e.g.:

```
<transition cond="yearOfBirth < 1990" target="offSales"/>
```

#### 4.2.4 Mapping ONFi specification to SCXML

Each ONFi state corresponds naturally to an SCXML state. Figures 4.5 illustrates. Some SCXML elements are not used, for example <parallel>.

##### **Sending information with events**

In theory events can include data, in a similar way to CSP channels. There are two methods of raising an event:

<event> which raises an event internally

<send> which sends an event to an external module (which can be another running SCXML interpreter)

It is not possible to include data using the first of these. Presumably this is because all states executing in parallel within the same SCXML interpreter have access to the same data structure (as mentioned

<b>T_RPP_ReadParams</b>	The target performs the following actions: <ol style="list-style-type: none"> <li>1. Request LUN <code>tLunSelected</code> clear SR[6] to zero.</li> <li>2. R/B# is cleared to zero.</li> <li>3. Request LUN <code>tLunSelected</code> make parameter page data available in page register.</li> <li>4. <code>tReturnState</code> set to T_RPP_ReadParams.</li> </ol>	
1. Read of page complete	→	<a href="#">T_RPP_Complete</a>
2. Command cycle 70h (Read Status) received	→	<a href="#">T_RS_Execute</a>
3. Read request received and <code>tbStatusOut</code> set to TRUE	→	<a href="#">T_Idle_Rd_Status</a>

```

<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"/>

```

SCXML states require an ID attribute: the state name at the top left of each description, being unique, fulfils this requirement. The actions above the list of transitions take place on entry to the state: therefore they correspond to actions within the `<onentry>` element.

<b>T_RPP_ReadParams</b>	The target performs the following actions: <ol style="list-style-type: none"> <li>1. Request LUN <code>tLunSelected</code> clear SR[6] to zero.</li> <li>2. R/B# is cleared to zero.</li> <li>3. Request LUN <code>tLunSelected</code> make parameter page data available in page register.</li> <li>4. <code>tReturnState</code> set to T_RPP_ReadParams.</li> </ol>	
1. <b>Read of page complete</b>	→	<a href="#">T_RPP_Complete</a>
2. Command cycle 70h (Read Status) received	→	<a href="#">T_RS_Execute</a>
3. Read request received and <code>tbStatusOut</code> set to TRUE	→	<a href="#">T_Idle_Rd_Status</a>

```

<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"/>

```

There is a one-to-one correspondence between ONFi transitions and SCXML `<transition>` elements. Any conditions can be represented with a `cond` attribute.

Figure 4.5: Mapping ONFi specification to SCXML: state T\_RPP\_ReadParams

in section 4.2.3) and therefore data transfer is unnecessary. This poses a problem for modelling the ONFi specification which often requires data to be included in events.

Since the SCXML is only an intermediate form, destined to be converted to CSP, we use CSP's channel notation directly within the SCXML, for example:

```
<event name="channel!data"> (for sending data with events)
and
<event name="channel?data"> (for receiving data with events)
```

#### 4.2.5 Apache Commons implementation

Several implementations exist, but the most popular and full-featured - and de-facto reference - interpreter for SCXML is that from Apache Commons [16]. The principal developer, Rahul Akolkar, is also closely involved with the W3C specification.

SCXML only specifies a state machine: it must be used in the context of a larger application. The Apache Commons fits this requirement by providing a Java library with external interfaces, which can then be integrated as necessary.

##### Direct use

The SCXML library's `.jar` file can be used directly on an SCXML document, for testing purposes. In this mode, the user is presented with the current state, and can manipulate the state machine by typing in events directly. The system then reacts appropriately, informing the user of state transitions taken and current data values.

Appendix A.2 shows a sample of the program's output, with user input in red.

##### Synchronization

As previously mentioned, states can react to events raised by other, parallel, states. This allows a rudimentary synchronization. However there is no way to force synchronization on the state that *raises* the event: it will not wait until the parallel state is waiting to receive an event.

This leads to unexpected deadlocks. Race conditions are not a problem however, since execution appears to be deterministic (it executes the same interleaving each time).

## 4.2.6 Adherence to SCXML standard

Because of this problem, along with the inability to send information along with events mentioned in section 4.2.4, it was decided not to adhere strictly to the SCXML standard for this project. This allows some CSP notation to be ‘mixed in’ with the SCXML.

The SCXML used is still well-formed XML: this is a requirement of the Saxon XSLT parser. However the Commons SCXML library will reject it.

This relaxation also allows the introduction of extra XML elements where required. These will be presented in the next section.

## 4.3 XML Transformations

Custom XSLT scripts were developed that transformed the same base SCXML file into CSP and also into an HTML format that was designed to be similar to the ONFi specification document. This made it simpler to check for implementation errors, since the two could be compared side-by-side.

### 4.3.1 SCXML to CSP

XSLT is mainly used to transform XML into XML (or to XML-like languages such as HTML) but is capable of producing text in almost any conceivable format.

One of the fundamental requirements of the language is that a compiler is not required to process the script from top to bottom, but instead has the flexibility to decide its own traversal route. This gives XSLT the flavour of a functional language. There is no variable reassignment, for example, though variables can be created and are accessible to all nodes of the tree below their declaration.

Therefore the following list of actions during conversion should not be considered to be a sequential ‘procedure’ as such. However the text of a  $CSP_M$  process definition is built up in identifiable stages.

- Channel names are derived from event names. Currently only single-event channels are supported. More complex channels must be created by hand, and therefore this stage can be suppressed using a `<csp:noGenerateChannels>` tag.
- For each state, a CSP process is generated, as follows:
  - The process name is generated from the state’s `id` attribute and capitalized, as per CSP convention.
  - Events in the `<onentry>` section are written sequentially, each followed by the  $CSP_M$  continuation token `->`.



- Processes may have a single `<if>` statement in the `<onentry>` section. If present, two forks are created, corresponding to the ‘true’ and ‘false’ outcomes of the tested expression.
- Each transition within the state is generated as follows:
  - \* The guard condition is written, performing any necessary conversions (e.g. `&&` becomes `and`).
  - \* If the transition is triggered by an event, that event is written followed by the  $CSP_M$  continuation token `->`.
  - \* The destination state (process) is written. Any state variables that were changed by an `<assign>` element, either in the state’s `<onentry>` element or in this transition, are changed here.

Transitions are separated by the  $CSP_M$  external choice token `[]`.

A ‘before and after’ example can be seen in Appendix A.

## Whitespace

$CSP_M$  is whitespace-dependent in certain circumstances. For example, definitions (processes, functions, channels, etc.) must start on a new line, though the definition itself may contain arbitrary whitespace including line feeds. There is no end-of-line character such as C’s semicolon.

XSLT includes whitespace only from text within text nodes<sup>1</sup>. Thus to ensure a line feed occurs in the desired location, the following XSLT snippet, which might on first glance appear superfluous, is required:

```
<xsl:text>
</xsl:text>
```

### 4.3.2 SCXML to HTML

This transformation is considerably simpler than the previous one. The script starts the html document by using the `<xsl:output>` tag to generate a DOCTYPE declaration. Following this, it creates the main `<html>` tag and below this, the `<head>` and `<body>` tags.

Into the `<head>` section goes a `<title>` and a link to the custom stylesheet `state.css`. This stylesheet formats the resulting output to match the ONFi document as closely as possible.

---

<sup>1</sup>In the XML Document Object Model, text that contains other XML elements is divided into nodes that contain only character data (called text nodes) interspersed with the other XML elements.

Into the `<body>` it inserts a `<div>` for each state. Below this are two tables, one for the ‘header’ — the state’s id and actions on entry — and one for the transitions, with one line per transition.

Some complications arise on entry to the state, since if there is only one action, it will not be numbered. Therefore the `<xsl:choose>` element decides whether or not to wrap the action(s) in a `<ol>` (ordered list) element.

If there is only one transition then the script generates the text ‘Unconditional’ to match the document.

The resulting HTML for three states, as rendered by Firefox, is shown in Figure 4.6. (`intCounter` is an internal state variable used at various places, for example when counting whether the correct number of address bytes has been received. It is not present in the original ONFi specification which does not descend into that level of detail.)

The HTML requires no final manual adjustments, and is valid according to the definition of XHTML v1.0 Strict. This can be verified at the W3C’s XHTML validation service [17].

### 4.3.3 Mandatory-only states

To speed up FDR2’s analysis, the optional commands and states were stripped from the SCXML, using XSLT.

An XML file was created with elements corresponding to the ID of the mandatory states. This file, along with the SCXML, is input to an XSLT transform script, which copies the original SCXML node-for-node. Any states, or transitions to states, without an element in this XML file are assumed to be optional and hence discarded. This transformation is analogous to a database ‘inner join’, but between two XML files rather than database tables.

The new SCXML file can be used as an input to the same XSLT that performs the SCXML to CSP conversion: no modification is required. In this way, new ‘mandatory-only’ CSP and HTML files were also generated, with a minimum of manual coding. Figure 4.7 illustrates the entire framework for the Target state machine. The same process is applied to those of the LUN and Host. GNU’s `make` utility was used to manage the resulting dependences: only dependent files are recompiled each time a change is made to the source.

## 4.4 Design of the host process

Though the ONFi specification does not provide a state machine description of the required host behaviour, the host is also implemented here using the same SCXML to CSP conversion method.

T_PP_LUN_DataPass	Event: tl_io.tLunSelected!dataBit
1. Unconditional	-> <a href="#">T_PP_LUN_DataWait</a>

T_PP_Cmd_Pass	Event: tl_cmd.tLunSelected!cmd
1. (if cmd==cmd11h)	-> <a href="#">T_PP_IlvWait</a>
2. (if cmd==cmd10h    cmd==cmd15h)	-> <a href="#">T_Idle</a>

T_PP_IlvWait	tReturnState set to T_PP_IlvWait
1. ht_ioCmd.cmd85h (if tCopyback==true) • intCounter set to 0	-> <a href="#">T_PP_AddrWait</a>
2. ht_ioCmd.cmd80h (if tCopyback==false) • intCounter set to 0	-> <a href="#">T_PP_AddrWait</a>
3. ht_ioCmd.cmd70h	-> <a href="#">T_RS_Execute</a>
4. ht_ioCmd.cmd78h • intCounter set to 0	-> <a href="#">T_RSE_Execute</a>
5. ht_read (if tbStatusOut==true)	-> <a href="#">T_Idle_Rd_Status</a>

The original ONFi descriptions are also presented for comparison:

T_PP_LUN_DataPass	Pass data byte/word received from host to LUN tLunSelected
1. Unconditional	→ <a href="#">T_PP_LUN_DataWait</a>

T_PP_Cmd_Pass	Pass command received to LUN tLunSelected
1. Command passed was 11h	→ <a href="#">T_PP_IlvWait</a>
2. Command passed was 10h or 15h	→ <a href="#">T_Idle</a>

T_PP_IlvWait	Wait for next Program to be issued. tReturnState set to T_PP_IlvWait.
1. Command cycle of 85h received <sup>1</sup> and tCopyback set to TRUE	→ <a href="#">T_PP_AddrWait</a>
2. Command cycle of 80h received <sup>1</sup> and tCopyback set to FALSE	→ <a href="#">T_PP_AddrWait</a>
3. Command cycle of 70h received	→ <a href="#">T_RS_Execute</a>
4. Command cycle of 78h received	→ <a href="#">T_RSE_Execute</a>
5. Read request received and tbStatusOut set to TRUE	→ <a href="#">T_Idle_Rd_Status</a>

Figure 4.6: Rendered HTML for three target states.

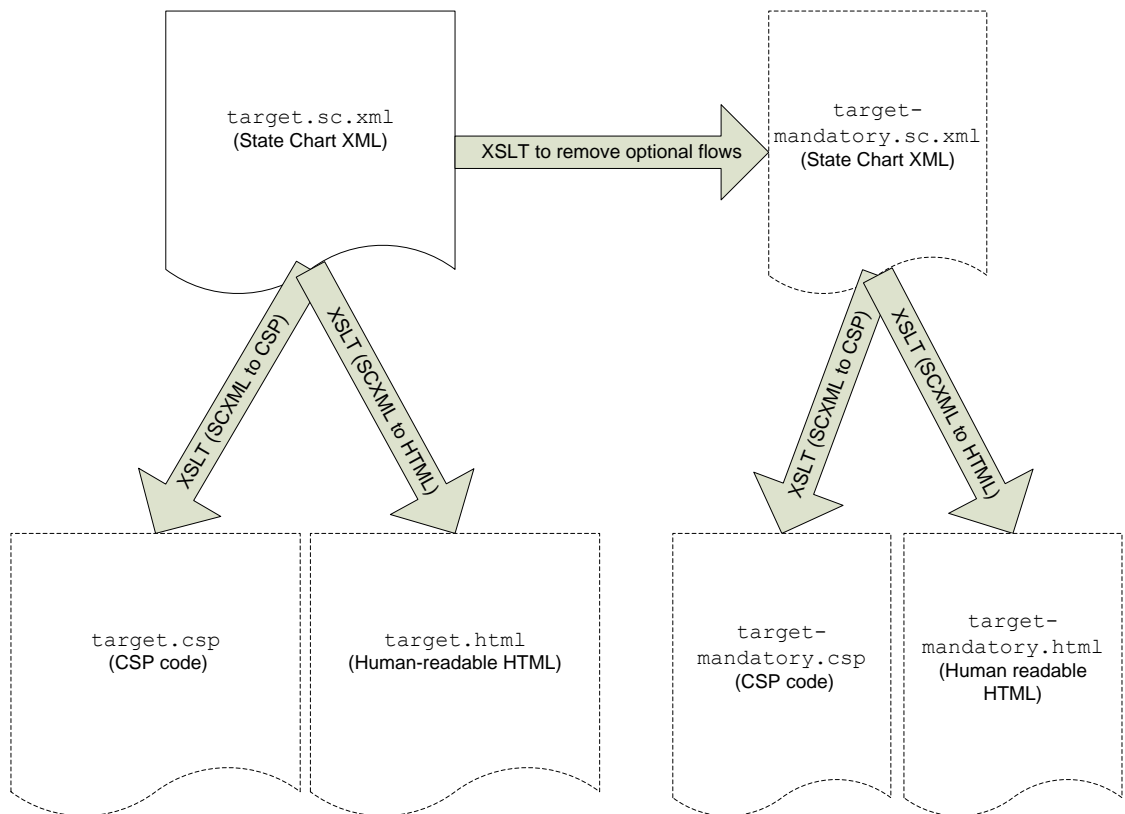


Figure 4.7: XSLT transformations of the Target state machine. Files with dotted lines are generated automatically

Some transitions have been omitted for clarity. Dotted lines indicate optional commands.

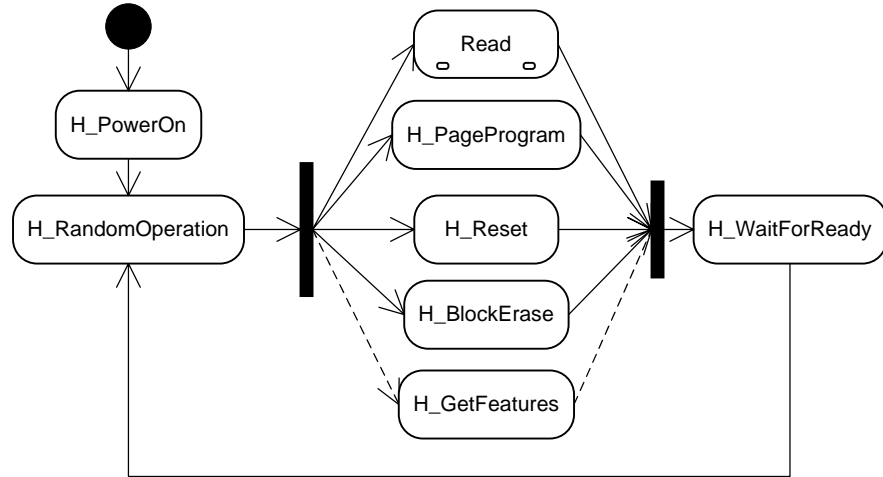


Figure 4.8: State diagram for the host process.

As envisioned in section 3.1.1, there are two separate host implementations, the first of which uses Read Status (the software host) and the other which uses the Ready / Busy pin (the hardware host).

The host's behaviour is quite simple. It performs the required reset at power on, then continuously loops, performing an arbitrary command each time. After each command, it waits until the target has returned to the ready state. This is illustrated in Figure 4.8.

#### 4.4.1 Read Command

The read command for a software host requires the use of Read Status. Depending on the length of time taken for the flash array to return the data, the host might receive 'busy' several times. The host's state machine therefore remains in the H\_ReadWait state, continuously polling the target until it receives a 'ready' result. At that point it completes the read by resending the 00h command, as required by the ONFi specification on p57. Figure 4.9 illustrates this process.

#### 4.4.2 Interleaved actions

To check that it is possible to read from multiple LUNs simultaneously, we create a special H\_MultiRead process. This sends a Read command to LUN 0, then without waiting for it to complete, also sends a Read to LUN 1. After this, it waits to read the results, using Read Status Enhanced to determine each LUN's readiness. It can read the results back in either order, i.e. first LUN 0 then LUN 1, or

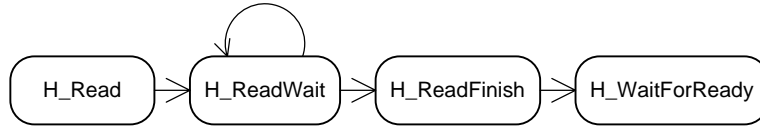


Figure 4.9: State diagram for the host's Read process.

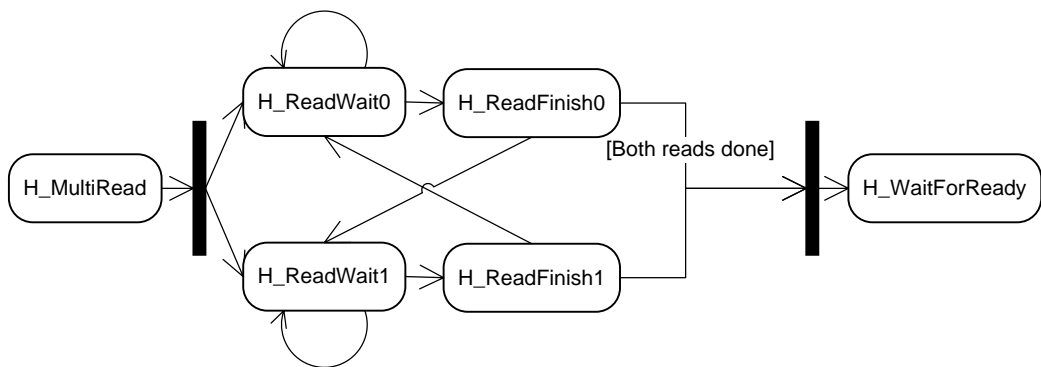


Figure 4.10: State diagram for the host's MultiRead process: testing simultaneous reads on two LUNs

vice versa.

The process is illustrated in Figure 4.10. The final `H_WaitForReady` is not strictly necessary, since by that stage the host has already been ascertained, using Read Status Enhanced, that both LUNs are ready.

### 4.4.3 Controlling the host

Each of the transitions from `H_RandomOperation` in Figure 4.8 is triggered by a custom event on the `eh` channel. If this channel is unsynchronized, then the host can randomly perform any event.

To 'direct' the host to perform a specific sequence of actions one can simply write a small CSP process that performs those actions and synchronizes with the host on the `eh` channel. For example, to perform the sequence of commands *PageProgram* → *Read* → *Reset*, the process would be as follows:

```
eh.h_PageProgram -> eh.h_Read -> eh.h_Reset
```

This method spares the labour of specifying the full host command sequence which would require the correct sequencing of address bits, command bytes, and read/write instructions.

## 4.5 Additional processes

To model the behaviour of a flash device fully, several processes were necessary in addition to *HOST*, *TARGET*, and *LUN*. These processes are simpler than those derived directly from the ONFi state machine, and so were written in CSP directly rather than via SCXML.

### 4.5.1 Status Register bit 6

This bit, present on every LUN, is linked to the value of the target's Ready/Busy pin, as explained in section 2.5.4. The LUN state machine generally denotes changes to this bit with text such as:

lunStatus[6] is set to one. lunStatus[6] value is indicated to the Target.

This implies a channel between target and LUN processes, which we shall call `t1_SR6`, following the convention in section 3.1.2. The second sentence of the above corresponds to a CSP event `t1_SR6.lun0!true` [where 'one' is modelled as `true`].

The specification is somewhat inconsistent, however, in its treatment of the target's reaction to these `t1_SR6` events. In some states, the target reacts to the SR[6] change with a state transition. For example, in the `T_Idle_Rd` state presented in Figure 4.14, one of the transitions (#2) explicitly mentions the SR[6] change. One might be tempted to model this as an event, `t1_SR6?lunID?sr6value`, and force synchronization on this channel at all times.

However this leads us into deadlock problems when the target is not in such an idle state, but the LUN still wishes to indicate a change to its SR[6] value. For example, the `T_PP_Execute` state, during its entry sequence, requests that all LUNs clear their page registers. This request triggers each LUN to move through the state cycle

`L_Idle` → `L_Idle_TargetRequest` → `L_Idle_ClearPageReg` → `L_Idle`.

On its return to `L_Idle`, the LUN sets its SR[6] value to one and indicates this to the target. This event, `t1_SR6.lun0!true`, must also be performed by the target. However the target's next interaction with the LUN (after receiving the address bytes from the host) is the Program command of `T_PP_LUN_Execute`. The target is not expecting a SR[6] update, and has no appropriate transition. Therefore the *TARGET* || *LUN* process will deadlock at this point. This deadlock is illustrated in Table 4.1.

#### **Solution 1 - Sampling SR[6] channel within Target process**

A footnote to the state `T_Idle_RB_Transition` says:


Target		LUN
→ 80h (Page Program) received		
Requests LUN clear page register	→	Clears page register
→ Address bytes received		
Issues Program command with associated address to LUN		Indicates lunStatus[6] value to Target

Table 4.1: SR[6] deadlock


Target		LUN
→ 70h (Read Status) received		Read completes
Sets tbStatus78hReq to FALSE		Sets lunStatus[6:5] to 11b
Sets tbStatusOut to TRUE		
Indicates 70h command to LUN		Indicates lunStatus[6] value to Target

Table 4.2: Sampling SR[6] within target: deadlock remains

R/B# may transition to a new value prior to the Target re-entering an idle condition when LUN level commands are in the process of being issued

Therefore it seems reasonable as a solution simply to add `t1_SR6` events to the appropriate places within the target state machine. This allows the required synchronization to take place and so the processes can continue. Finding the appropriate places is a matter of trial and error, however FDR's deadlock checker allows this to be done without much difficulty.

There are, however, still problems in certain places. Consider when the target has finished sending a 00h read command to the LUN, and is now waiting for the command to complete. The LUN is in the state `L_Idle_Rd`. The target is in the state `T_Idle_Rd`. The sequence of events in Table 4.2 can then occur, culminating in deadlock.

It cannot be resolved by adding a `t1_SR6` event to the target machine, because in other event sequences, the read does not complete on the LUN and thus the LUN will not attempt to signal on the `t1_SR6` channel.

## Solution 2 - Separate `READYBUSY` process

The authors of the ONFi specification stated [18]:

... that R/B# is a hardware AND of all SR[6] values and that it will change when there is a change with a particular LUN regardless of the Target state machine context.



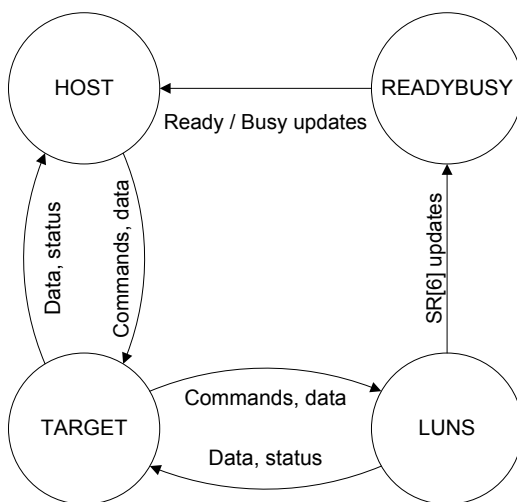


Figure 4.11: Parallel CSP processes of the entire system

Given that it is implemented in hardware, and therefore independent of the target's state, the other approach was to model the Ready/Busy pin as a separate CSP process, *READYBUSY*. It synchronizes with the LUN's  $\tau 1\_SR6$  events (the target no longer synchronizes on this channel), and invokes a *ht\_readyBusy* event upon every change in SR[6]. The *host* is still synchronized on the *ht\_readyBusy* channel, even though the target is not.

The total system now has four parallel processes:

*HOST* || *TARGET* || *LUNS* || *READYBUSY*

Figure 4.11 illustrates.

Though the target is no longer reacting to SR[6] events, it must still be aware of the value of Ready/Busy during certain states. For example, during a Page Program, if Ready/Busy is false (i.e. busy), then the target sets *tbStatus78hReq* to true. This is because there are now multiple interleaved operations taking place (since the host did not wait for Ready/Busy to return to ready before issuing the Page Program). A Read Status Enhanced (78h) command must now be issued before a read can take place.

The solution is to introduce some extra events in the target state machine. Recall that in some idle states, the target has transitions that are triggered by SR[6] changes (in all other states, the target does not react).

We 'garnish' these states with extra events at their entry and exit points, that signal to the *READYBUSY* process that the target now requires notification of any SR[6] changes. These entry and exit signals cause the *READYBUSY* to transition to a different process, *READYBUSY\_PASSTHROUGH*.

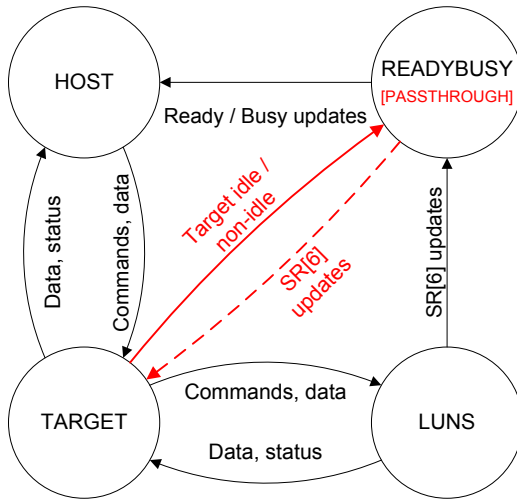


Figure 4.12: Parallel CSP processes of the entire system, modified

This process has extra events,  $\text{tr\_SR6}$ , which are synchronized with the target, and occur after  $\text{SR}[6]$  changes.

The net effect is that the target, when in an idle state, receives notification of changes to  $\text{SR}[6]$  and so can update its *readyBusy* state variable appropriately. This modified system is shown in Figure 4.12.

#### 4.5.2 LUN ‘innards’ process

An aim of the project was to model the unpredictability of the flash array: i.e. variation in transfer times, and arbitrary failures.

The LUN state machine, at several points, waits for flash operations to complete. During the wait, it is still capable of responding to status requests from the host: it is not blocked. This can be seen clearly in the state  $\text{L\_RD\_ArrayRead}$ , shown in Figure 4.13.

Transition 1 occurs upon completion of the read. This event is not seen by the target, which is only capable of synchronizing on transition 2.

The XSLT conversion process always results in external choice between transitions. It is necessary to introduce *internal* choice to the model to represent the possibility of nondeterministic outcomes after the same sequence of external events.

The simplest way to model this nondeterminism would be a process along these lines:

$$\text{FLASH} \hat{=} \text{read} \rightarrow \text{FLASH\_WORKING}$$

L_RD_ArrayRead	The LUN performs the following actions:		
	<ol style="list-style-type: none"> <li>1. lunStatus[6:5] is cleared to 00b.</li> <li>2. lunStatus[6] value is indicated to the Target.</li> <li>3. lunLastConfirm set to last command cycle (30h or 35h).</li> <li>4. Read the requested page from the array.</li> <li>5. lunReturnState set to L_RD_ArrayRead.</li> </ol>		
	1. Read of requested page complete	→	<a href="#">L_RD_Complete</a>
	2. Target requests status or status command received	→	<a href="#">L_Status_Execute</a>

Figure 4.13: LUN state L\_RD\_ArrayRead

$$FLASH\_WORKING \hat{=} busy \rightarrow FLASH\_WORKING \sqcap done \rightarrow FLASH$$

where the *LUN* process initiates the *read* event, then waits to synchronize on the *done* event.

Unfortunately, upon hiding the *busy* event (since it is internal to the *FLASH* process), livelock results. This is because the *FLASH\_WORKING* process can perform an infinite number of *busy* events, without ever choosing *done*.

To avoid this, on each internal choice the array makes at least some progress towards completion. This requires the process to store some state to represent progress made so far. In this example, we add either 1 or 2 to the *progress* variable, and generate a *done* event when *progress* reaches 10.

$$FLASH\_WORKING(progress) \hat{=} \left. \begin{array}{l} FLASH\_WORKING(progress + 1) \\ \sqcap \\ FLASH\_WORKING(progress + 2) \\ \sqcap \\ done \rightarrow FLASH \end{array} \right\} \begin{array}{l} \text{if } progress < 10 \\ \\ \text{otherwise} \end{array}$$

This process is put in parallel with the LUN state machine process. The actual implementation *LUN\_INNARDS* is somewhat more complicated due to the different types of flash operation (Read, Page Program, Erase, Reset) and the possibility that an operation may be interrupted by a reset command.

In addition, it is necessary to modify the L\_BE\_Erase\_Wait (and other similar commands in which the LUN waits for internal events to complete) as follows:

The LUN must ‘poll’ the *LUN\_INNARDS* process on *entry* to the state. If the innards is about perform *done*, the ‘read status’ branch is disallowed. Otherwise livelock can occur, when the innards process is in fact ready, but the external choice (of the LUN) between responding to the target, and

T_Idle_Rd	Wait for read request (data or status) or other action. tReturnState set to T_Idle_Rd.		
1. WP# signal transitioned	→		<a href="#">T_Idle_WP_Transition</a>
2. LUN indicates its SR[6] value transitioned	→		<a href="#">T_Idle_RB_Transition</a>
3. Read request received and tbStatusOut set to TRUE	→		<a href="#">T_Idle_Rd_Status</a>
4. Read request received and (tLastCmd set to 90h or EEh)	→		<a href="#">T_Idle_Rd_XferByte</a>
5. Read request received and (tLastCmd set to ECh or EDh)	→		<a href="#">T_Idle_Rd_LunByte</a>
6. Read request received and tbStatus78hReq set to FALSE <sup>1</sup>	→		<a href="#">T_Idle_Rd_LunData</a>
7. Command cycle 05h (Change Read Column) received and tbChgCol set to TRUE	→		<a href="#">T_CR_Execute</a>
8. Command cycle of 31h received and tbStatus78hReq set to FALSE	→		<a href="#">T_Idle_Rd_CacheCmd</a>
9. Command cycle of 3Fh received and tLastCmd set to 31h and tbStatus78hReq set to FALSE	→		<a href="#">T_Idle_Rd_CacheCmd</a>
10. Command cycle received	→		<a href="#">T_Cmd_Decode</a>
NOTE: 1. When tbStatus78hReq is set to TRUE, a Read Status Enhanced (78h) command followed by a 00h command shall be issued by the host prior to reading data from a particular LUN.			

Figure 4.14: Target state T\_Idle\_Read

responding to the innards, always chooses to do the Read Status (and hence always returns ‘not ready’).

## 4.6 State Transitions

As described in 2.5.5, the transitions from each state are presented in order of priority. This priority ordering results in slightly different semantics from CSP’s external choice mechanism, since the descriptions of the transitions need not be mutually exclusive. An example should make this clear. Consider the state shown in Figure 4.14.

Here transition no. 3 will trigger if tbStatusOut is set to TRUE and a read request is received. In CSP<sub>M</sub> this is written (using the & guard notation) as:

```
tbStatusOut==true & ht_read -> T_IDLE_RD_STATUS
```

If the text for transition no. 4 is taken literally, one might write:

```
(tLastCmd==cmd90h or tLastCmd==cmdEEh) & ht_read -> T_IDLE_RD_XFERBYTE
```

but this neglects the possibility that `tbStatusOut` might be `TRUE`, in which case transition 3 is in fact the appropriate choice.

Thus it is necessary to consider every higher priority transition before writing the correct guard expression. In this case it is sufficient to write:

```
(tbStatusOut==false and (tLastCmd==cmd90h or tLastCmd==cmdEEh))
  & ht_read -> T_IDLE_RD_XFERBYTE}
```

By transition 6 the correct guard expression has become rather complicated because of this requirement to exclude all the previous transition possibilities:

```
(tbStatusOut==false and tbStatus78hReq==false
and tLastCmd!=cmd90h and tLastCmd!=cmdEEh
and tLastCmd!=cmdECh and tLastCmd!=cmdEDh)
  & ht_read -> T_IDLE_RD_LUNDATA
```

## 4.7 Testing

As well as checking the composite system for deadlock and livelock, we adopt the two other approaches described in section 3.2.3: making the host repeatedly perform arbitrary commands; and checking that certain [un]expected sequences of events are [in]correct refinements.

### 4.7.1 Arbitrary commands

The `eh` channel (described in section 4.4.3) supports events from a CSP datatype called `HostEvent`. Since some commands are optional, we create a set `host_events_mandatory` that excludes them.

The process to ‘drive’ the host simply picks one command, nondeterministically:

```
ANYCMD_SW = |~| e : host_events_mandatory @ eh.e -> ANYCMD_SW
```

This process runs in parallel with the host / target / LUNs combination:

```
HOST_SW_ANYCMD = HOST_SW_TARGET_TWOLUNS [| {|eh|} |] ANYCMD_SW
```

It is checked for deadlock, as follows:

```
assert HOST_SW_TARGET_TWOLUNS :[deadlock free [F] ]
```

Deadlock-freedom is not enough, however. The system must eventually perform the command chosen by the `ANYCMD_SW` process. All events are hidden, except for those chosen:

Channel	Value	Comment
ht_ioCmd	60h	Start of block erase command
-	-	...intervening address bits are hidden ...
ht_ioCmd	D0h	End of block erase command
ht_ioCmd	70h	Read Status
ht_ioDataOut	false	Result of Read Status: <i>busy</i>
ht_ioCmd	70h	Read Status
ht_ioDataOut	true	Result of Read Status: <i>ready</i>

Table 4.3: Expected events for a Block Erase

```
HOST_SW_ANYCMD_HIDDEN = HOST_SW_ANYCMD \ diff(Events, {|eh|})
```

and check for livelock:

```
assert HOST_SW_ANYCMD_HIDDEN : [livelock free [FD] ]
```

## 4.7.2 Refinements

Each of the refinements follows a similar pattern. The host / target / LUN system is the specification. It is not necessary for the refinement to include every single event; therefore they are restricted to commands (i.e. events on the `ht_ioCmd` channel, as described in section 3.1.2) and data input and output (`ht_ioDataIn` and `ht_ioDataOut` respectively). All other events are hidden from the specification.

### Block Erase

The simplest command (page 53 of the ONFi specification), its expected events are shown in Table 4.3. Since the address events are hidden, all we will see are the command bytes, followed by Read Status. Note that the *busy* result, here shaded in grey, might happen zero, one, or more times, depending on how long the flash erase operation takes.

### Read

The read command (page 57) has two parts:

- the initial command including the address, followed by
- read(s) from the page register (once the LUN has finished transferring the data from the flash array).

Channel	Value	Comment
ht_ioCmd	00h	Start of read command
-	-	...intervening address bits are hidden ...
ht_ioCmd	30h	End of read command
ht_ioCmd	70h	Read Status
ht_ioDataOut	false	Result of Read Status: <i>busy</i>
ht_ioCmd	70h	Read Status
ht_ioDataOut	true	Result of Read Status: <i>ready</i>
ht_ioCmd	00h	Complete read
ht_ioDataOut	(data)	Data read
ht_ioCmd	70h	Read Status
ht_ioDataOut	true	Result of Read Status: <i>ready</i>

Table 4.4: Expected events for a Read

```

-- check whether Read Status can get both Ready or Busy after a read
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_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 -> STOP
assert READ_SPEC [T= READ_IMPL0
READ_IMPL1 = POWERON; ht_ioCmd.cmd00h -> ht_ioCmd.cmd30h
-> ht_ioCmd.cmd70h -> ht_ioDataOut.false -- read status returned busy, so wait
-> ht_ioCmd.cmd70h -> ht_ioDataOut.true -- read status returned ready, so read
-> ht_ioCmd.cmd00h -> ht_ioDataOut.false
-> ht_ioCmd.cmd70h -> ht_ioDataOut.true -> STOP
assert READ_SPEC [T= READ_IMPL1

```

Figure 4.15: CSP for Read refinement check

The second part can only proceed when the LUN has returned to the ready state. The host should be acting according to Figure 4.9, so one expects to see the sequence of events shown in Table 4.4. The last Read Status is not strictly necessary (the LUN is always *ready* by then) but is included because all the host operations have the same basic structure illustrated in Figure 4.8, which always includes a final Read Status.

The CSP for this check is shown in Figure 4.15. `READ_IMPL0` is a trace in which Read Status immediately returns *ready*, while `READ_IMPL1` is a trace in which Read Status returns *busy* the first time, followed by *ready*.

Channel	Value	Comment
ht_ioCmd	80h	Start of page program command
-	-	...intervening address bits are hidden ...
ht_ioDataIn	(data)	Data written
ht_ioCmd	10h	End of page program command
ht_ioCmd	70h	Read Status
ht_ioDataOut	false	Result of Read Status: <i>busy</i>
ht_ioCmd	70h	Read Status
ht_ioDataOut	true	Result of Read Status: <i>ready</i>

Table 4.5: Expected events for a Page Program

### Page Program

This command is simpler, since the data can be written immediately to the page register by the target. The events are shown in Table 4.5.

### 4.7.3 Non-refinements

The following sequences of events should not be seen, so are a useful check that the model is behaving as expected.

**Block Erase followed by Read Status Enhanced** - the host, as programmed here, only uses Read Status Enhanced during the MultiRead.

**Read without Read Status** - this refinement has the host immediately attempt a read, without the intervening Read Status, and should be impossible.

**Multiple Reads completed followed by *busy*** - once the host has completed both reads, both LUNs are in the ready state, so a Read Status Enhanced cannot return *busy*.



# Chapter 5

## Evaluation

### 5.1 Verification of ONFi specification

Various anomalies were discovered, and are summarised in the following sections.

#### 5.1.1 Ready / Busy

The problems encountered with this pin were summarised in section 4.5.1. The solution was only a partial one, since the specification calls in several places for Ready / Busy to be set by the target, which is impossible if it is a hardware AND of the LUNs' SR[6] bits.

#### 5.1.2 Status Register Update

At the start of certain operations, the target requests that the LUN set its Status Register 6 bit to a certain value. The precise event sequence is shown in Table 5.1. Clearly the update operation has no effect, since SR[6] is reset to 1 when the LUN returns to L\_Idle.

In correspondence with ONFi (Appendix B), Michael Abraham of Micron suggested altering the T\_RPP\_ReadParams state to remove the request to set SR[6], and insert a SR[6] change into the LUN's

LUN state	Event	SR[6] value
L_Idle	Target request received	1
L_Idle_TargetRequest	Target requests SR register update	1
L_SR_Update	Update lunStatus as indicated by the Target	0
L_Idle	lunStatus[6] set to one	1

Table 5.1: Status Register Update: event sequence for setting SR[6] to 0


Target		LUN
→ ECh (Read Parameter Page) received		
Target requests LUN invalidate page register	→	LUN invalidates page register
→ Target receives 00h address cycle		
Target requests LUN clear SR[6] to zero	→	LUN clears SR[6] to zero
Target requests LUN make page parameter data available in page register	→	LUN starts reading parameter page into page register
→ 70h received (Read Status)		LUN finishes reading parameter page into page register
Target indicates 70h received to LUN		LUN indicates to target that parameter page data is in page register

Table 5.2: Read Parameter Page: deadlock 1

L\_Idle\_RdPp state. Similar changes need to be made to the other target states which request a SR[6] change (T\_RPP\_Complete, T\_RU\_ReadUid, T\_RU\_Complete, T\_SF\_Complete, T\_SF\_UpdateStatus, T\_GF\_RetrieveParams, and T\_GF\_Ready).

### 5.1.3 Read Parameter Page

There are several problems with this command. It was possible to modify the CSP model to solve some, but not all, of them.

#### Read Status

The CSP model deadlocks under the sequence of actions shown in Table 5.2.

There are analogous deadlocks in other sequences, specifically during the Target states T\_RST\_Perform, T\_RU\_ReadUid, T\_SF\_Complete, and T\_GF\_ReadParams. In all of these states, the LUN indicates directly to the target that a command has completed, rather than using SR[6]. In each case, this indication can take place while the Target is simultaneously attempting a Read Status, causing deadlock.

This can be solved by adding an additional  $\tau 1\_sync$  event to the first ('read complete') transition from the target state, and the same event to the LUN's first transition. This forces both the target and LUN along the correct transition paths: the target can no longer perform a Read Status while the LUN completes the read. However it is inconsistent with the implicit requirement that a host can perform a Read Status at any time to determine the device's readiness.


Target		LUN
→ ECh (Read Parameter Page) received		
Target requests LUN invalidate page register	→	LUN invalidates page register
→ Target receives 00h address cycle		
Target requests LUN clear SR[6] to zero	→	LUN clears SR[6] to zero
Target requests LUN make page parameter data available in page register	→	LUN starts reading parameter page into page register
→ 70h received (Read Status)		
tbStatusOut is set to TRUE		
Indicates 70h command to LUN	→	lunStatusCmd is set to 70h
Returns to T_RPP_ReadParams		Returns to L_Idle_RdPp
Requests LUN make parameter page data available		

Table 5.3: Read Parameter Page: deadlock 2

### Return to T\_RPP\_ReadParams after a Read Status

The state variable tReturnState is used in several locations. In general it is accessed after a Read Status (or Read Status Enhanced) to return the target to the appropriate state while it waits for a LUN operation to complete.

In the case of a Read Parameter Page operation, it is set to T\_RPP\_ReadParams. On return to this state after a Read Status, the target once again performs the actions at the beginning of the state.

These are to:

- request the LUN set its SR[6] to zero
- request the LUN make the parameter page data available in the page register

However, the LUN, being in the state L\_Idle\_RdPp, can respond to neither of these actions. Table 5.3 illustrates this sequence of events.

These deadlocks cannot be solved using additional synchronizing events.

### 5.1.4 Reset

#### Reset of more than one LUN

The Read Status command only returns the status of the most recently accessed LUN (tLunSelected): for a reset, it should be the status of both.


Target		LUN
→ FFh (Reset) received		
Target sends a Reset request to each LUN.	→	Performs reset of the LUN
		Reset of LUN complete
Target requests all LUNs invalidate page register		LUN indicates to target that reset is complete

Table 5.4: Reset: deadlock 1


Target		LUN
→ FFh (Reset) received		
Target sends a Reset request to each LUN.	→	Performs reset of the LUN
→ 70h received (Read Status)		Reset of LUN complete
Target indicates 70h received to LUN		LUN indicates to target that reset is complete

Table 5.5: Reset: deadlock 2

### Invalidate page register

The description of `T_RST_Execute` states that the target sends a reset request to each LUN, then requests that each LUN invalidate its page register(s). However the LUN cannot do this until the reset has completed and the LUN has returned to `L_Idle`. If the target waits for this, it will be blocked and unable to react to a Read Status as implied by the next Target state `T_RST_Perform`. In any case, the LUN’s next interaction with the target must be either a ‘reset complete’ or a ‘read status’, not an ‘invalidate page register’. This deadlock is illustrated in Table 5.4.

### Read Status

The model can deadlock when a software host attempts a Read Status after a reset. This is fundamentally the same as Read Parameter Page’s first deadlock, and is shown in Table 5.5.

## 5.2 Suitability of CSP for project

It is safe to say that to verify a specification as complex as ONFi’s by hand would be impossible. CSP has been used for examining state charts in the past, for example in [19]. The one-to-one correspondence between CSP processes and state machine states allowed for direct conversion, avoiding the need to abstract away too much detail.

### 5.2.1 Tractability

Unfortunately the full ONFi model proved too much for the FDR2 model-checker, failing to compile (as described in Appendix D). The deadlocks described above were discovered in the mandatory-only model. This ‘mandatory’ model was in fact extended beyond the strictly mandatory commands to include the optional Read Status Enhanced (78h), to allow analysis of multiple simultaneous LUN operations.

## 5.3 Success of automation

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
  - sets to differentiate mandatory from optional commands
- minor supplementary CSP processes (*LUN\_INNARDS*, *READYBUSY*)
- parallel composition of host / target / LUN state machines
- specification of deadlock/livelock checks

Ideally the `<send>` notation for passing data with events would be used — currently the project’s SCXML has CSP event notation ‘mixed in’ — and converted to CSP events. This would bring the project’s SCXML closer to adherence to the standard.

## 5.4 Host process limits

The host process, thanks to its limited design, is guaranteed to deliver only ‘sensible’ sequences of commands to the target. This means that deadlocks or other anomalies that occur during atypical command sequences will not be picked up. To assert that the ONFi specification has been verified, based on this project’s results, requires that the host process:

- is sufficiently complex to model all possible realistic interactions with the target, and

- is itself free of bugs and other implementation errors

While the latter can be assumed with some confidence due to the host's simple (perhaps simplistic) design, clearly there are realistic behaviours that have not been modelled, for example:

- reading one LUN while programming another
- interrupting a command with a reset (FFh) command
- changing the read column (05h)

In addition, the host has not been extended to use any of the optional commands (e.g. copyback reads, interleaving at the LUN level).

# Chapter 6

## Conclusions

The vision of a fully verified file system is a step closer to realization. The software part of a persistent storage system must be matched by equally dependable hardware: the ONFi specification, with at most minor modifications, can credibly be put forward for this role.

The future of formal verification may well lie in the approach taken in this project; that is, automatic or semiautomatic conversion of human-readable specifications into machine-readable formats such as CSP. One of the reasons given for resistance to widespread adoption of formal methods is the unfamiliar notation. A base XML format that can be easily converted to a more digestible representation would help mitigate this problem.

### 6.1 Future work

#### 6.1.1 Optional commands

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. FDR2 has the ability to compress the state space during refinement checking. This was briefly explored during the project but the results were inconclusive. A more thorough investigation of this functionality may remedy the state-space problems mentioned.

#### 6.1.2 ONFi version 2.0

ONFi have since released version 2.0 of the specification. The state machine has not changed dramatically: it could be modelled in the same framework without much difficulty.

### 6.1.3 Extending and streamlining the SCXML to CSP conversion

The conversion of SCXML to CSP would benefit from further automation, to reduce the necessary manual interventions described in section 5.3. Ideally it would be possible to convert arbitrary SCXML into CSP and then analyse it, without the in-depth knowledge of CSP that is currently required. This would be a substantial undertaking. It would also require FDR2's explanations of deadlocks and other refinement problems to be automatically linked back to the source SCXML and presented to the user, who would then never have to interpret the raw CSP. This could potentially make use of FDR2's TCL scripting mode, which allows direct use of the model-checking engine, without invoking the X11 front-end.

### 6.1.4 Integration with a verified filesystem

As previously mentioned, a verified flash memory store is only one part of the grander vision of Dependable Systems Evolution. In theory a system that is composed of verified subsystems will itself display predictable behaviour and will be easily analysed. However composing the subsystems is no easy task, and requires that they present well-defined interfaces at their points of interaction. Verified filesystems have been the subject of prior work, for example in [9]. A starting point for future work could be to clarify:

- physical storage services required by the above file system
- services offered by the ONFi specification

Ideally one would mirror the other; realistically there may be enough overlap to develop a formal model that incorporates both.



# Bibliography

- [1] A. Butterfield and J. Woodcock, “Formalising flash memory: First steps,” in *12th IEEE International Conference on Engineering Complex Computer Systems*, 2007, pp. 605–610. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1271068>
- [2] A. Hall, “Seven myths of formal methods,” *IEEE Softw.*, vol. 7, no. 5, pp. 11–19, 1990.
- [3] An analysis of two formal methods: Comparison of tool support. [Online]. Available: <https://www.dacs.dtic.mil/techs/2fmethods/tools.php>
- [4] C. A. R. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
- [5] P. Welch and N. Brown. Communicating Sequential Processes for Java (JCSP). [Online]. Available: <http://www.cs.kent.ac.uk/projects/ofa/jcsp/>
- [6] T. Hoare and R. Milner, Eds., *Grand Challenges in Computing Research*. The British Computer Society, 2004.
- [7] J. P. Bowen, “Formal methods in safety-critical standards,” in *Proc. 1993 Software Engineering Standards Symposium (SESS’93), Brighton, UK*. IEEE Computer Society Press, 30 August – 3 September 1993, pp. 168–177. [Online]. Available: <http://citeseer.ist.psu.edu/article/bowen93formal.html>
- [8] J. Woodcock. (2003, May) Dependable Systems Evolution - A Grand Challenge for Computer Science. [Online]. Available: [http://www.nesc.ac.uk/esi/events/Grand\\_Challenges/proposals/dse.pdf](http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/dse.pdf)
- [9] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard, “On Verifying a File System Implementation,” Tech. Rep., 2004.

- [10] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, “Using model checking to find serious file system errors,” in *OSDI’04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 19–19.
- [11] R. Joshi and G. J. Holzmann, “A mini challenge: build a verifiable filesystem,” *Form. Asp. Comput.*, vol. 19, no. 2, pp. 269–272, 2007.
- [12] ONFi website. [Online]. Available: <http://www.onfi.org>
- [13] C. Edwards, “The Dash For Flash,” *IEEE Engineering and Technology*, Feb 2008.
- [14] D. Harel, “Statecharts: A visual formalism for complex systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.
- [15] World Wide Web Consortium (W3C). State Chart XML (SCXML): State Machine Notation for Control Abstraction. [Online]. Available: <http://www.w3.org/TR/scxml/>
- [16] T. A. Project. Commons SCXML. [Online]. Available: <http://commons.apache.org/scxml/>
- [17] W3C Markup Validation Service. [Online]. Available: <http://validator.w3.org>
- [18] A. Huffman, personal communication.
- [19] W. L. Yeung, K. R. P. H. Leung, J. Wang, and W. Dong, “Modelling and model checking suspendible business processes via statechart diagrams and CSP,” *Sci. Comput. Program.*, vol. 65, no. 1, pp. 14–29, 2007.

# Appendix A

## Example State Chart XML

### A.1 Lift

This example presents a model of a lift in a building with five floors<sup>1</sup>. It responds to *up* and *down* events from the user. Doors are opened and closed with the *open* and *close* events.

Some restrictions are necessary to ensure realistic operation:

- The lift may not go above floor 5 nor below floor 1.
- When the doors are open, the lift will not move. The only possible event is *close*.

#### A.1.1 SCXML representation

Figure A.1 shows the SCXML for the above example. The `Data(liftState,...)` wrapper around the data items is necessary in the Apache Commons implementation, since it uses the JEXL expression language to process these expressions. Note that `<` and `>` must be escaped as `&gt;` and `&lt;` in XML.

#### A.1.2 Direct use of Commons SCXML

The output in Figure A.2 shows what happens when the SCXML is used directly with the Commons SCXML library.

#### A.1.3 CSP representation

The SCXML to CSP converter is called from the command line as illustrated in Figure A.3.

---

<sup>1</sup>This example is adapted from lecture notes prepared by Dr Andrew Butterfield.

```

<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
  initialstate="lift">
  <datamodel>
    <data name="liftState">
      <floor xmlns="">1</floor>
      <doorsOpen xmlns="">>false</doorsOpen>
    </data>
  </datamodel>
  <state id="lift">
    <onentry>
      <log label="lift_at_floor" expr="Data(liftState,'floor')"/>
    </onentry>
    <transition event="up" target="lift"
      cond="Data(liftState,'doorsOpen')==false & &
        Data(liftState,'floor') &lt; 5">
      <assign location="Data(liftState,'floor') "
        expr="Data(liftState,'floor') + 1"/>
    </transition>
    <transition event="down" target="lift"
      cond="Data(liftState,'doorsOpen')==false & &
        Data(liftState,'floor') &gt; 1">
      <assign location="Data(liftState,'floor') "
        expr="Data(liftState,'floor') - 1"/>
    </transition>
    <transition event="open" target="lift"
      cond="Data(liftState,'doorsOpen')==false">
      <assign location="Data(liftState,'doorsOpen') " expr="true"/>
    </transition>
    <transition event="close" target="lift"
      cond="Data(liftState,'doorsOpen')==true">
      <assign location="Data(liftState,'doorsOpen') " expr="false"/>
    </transition>
  </state>
</scxml>

```

Figure A.1: SCXML for Lift example

```

20-Jul-2008 16:43:45 org.apache.commons.scxml.model.Log execute
INFO: lift_at_floor: 1.0
20-Jul-2008 16:43:45 org.apache.commons.scxml.env.SimpleSCXMLListener onEntry
INFO: /lift

up

20-Jul-2008 16:43:51 org.apache.commons.scxml.env.SimpleSCXMLListener onExit
INFO: /lift
20-Jul-2008 16:43:51 org.apache.commons.scxml.env.SimpleSCXMLListener onTransition
INFO: transition (event = up, cond = Data(liftState,'doorsOpen')==false
      && Data(liftState,'floor')<5, from = /lift, to = /lift)
20-Jul-2008 16:43:51 org.apache.commons.scxml.model.Log execute
INFO: lift_at_floor: 2.0
20-Jul-2008 16:43:51 org.apache.commons.scxml.env.SimpleSCXMLListener onEntry
INFO: /lift

down

20-Jul-2008 16:43:56 org.apache.commons.scxml.env.SimpleSCXMLListener onExit
INFO: /lift
20-Jul-2008 16:43:56 org.apache.commons.scxml.env.SimpleSCXMLListener onTransition
INFO: transition (event = down, cond = Data(liftState,'doorsOpen')==false
      && Data(liftState,'floor')>1, from = /lift, to = /lift)
20-Jul-2008 16:43:56 org.apache.commons.scxml.model.Log execute
INFO: lift_at_floor: 1.0
20-Jul-2008 16:43:56 org.apache.commons.scxml.env.SimpleSCXMLListener onEntry
INFO: /lift

```

Figure A.2: Using the Apache Commons SCXML library directly with the Lift example

```

java -jar saxon/saxon9.jar -t scxmlTests/lift.sc.xml SCXMLtoCSP.xslt JEXL=yes

Saxon 9.0.0.6J from Saxonica
Java version 1.6.0_06
Stylesheet compilation time: 1693 milliseconds
Processing file:/home/art/dissertation/scxmlTests/lift.sc.xml
Building tree for file:/home/art/dissertation/scxmlTests/lift.sc.xml using class
net.sf.saxon.tinytree.TinyBuilder
Tree built in 14 milliseconds
Tree size: 24 nodes, 6 characters, 32 attributes

```

Figure A.3: Calling the Saxon XSLT processor

```

channel up
channel down
channel open
channel close
INITIAL_LIFT=LIFT(1,false)
LIFT(floor,doorsOpen) =
  ( (doorsOpen==false and floor<5)
    & (up -> LIFT(floor + 1,doorsOpen))
  []
  (doorsOpen==false and floor>1)
    & (down -> LIFT(floor - 1,doorsOpen))
  []
  (doorsOpen==false)
    & (open -> LIFT(floor,true))
  []
  (doorsOpen==true)
    & (close -> LIFT(floor,false))

```

Figure A.4: CSP for Lift example

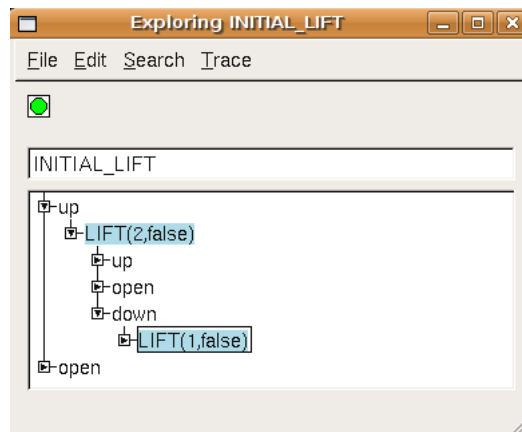


Figure A.5: Exploring the CSP process using Probe

The optional `JEXL=yes` command line parameter informs the XSLT script that it must remove the `Data('...')` from around the data items before converting them to CSP.

It yields the CSP shown in Figure A.4. The channels have been generated automatically from the set of events that exist within the SCXML. Since default values are provided for the data, the XSLT also creates an initial state with an `INITIAL_` prefix and fills in those defaults.

The Probe tool, available from the makers of FDR2, allows the user to explore a CSP process directly. Figure A.5 shows the Probe tool performing the same sequence of events as in Figure A.2.

```

<state id="T_RPP_ReadParams">
  <onentry>
    <event name="t1.tLunSelected!targRequest" />
    <event name="t1_setSR6.tLunSelected!false" />
    <assign location="isReadyBusy" expr="false" />
    <event name="t1.tLunSelected!targRequest" />
    <event name="t1.tLunSelected!retrieveParameters" />
    <assign location="tReturnState" expr="T_RPP_ReadParams" />
  </onentry>
  <transition event="t1.tLunSelected.readPageComplete" target="T_RPP_Complete" />
  <transition event="ht_ioCmd.cmd70h" target="T_RS_Execute" />
  <transition event="ht_read" cond="tbStatusOut==true" target="T_Idle_Rd_Status" />
</state>

```

```

T_RPP_READPARAMS(tbStatusOut, tbChgCol, tCopyback, tLunSelected, tLastCmd, tReturnState,
tbStatus78hReq, cmd, isReadyBusy, isWriteProtected, dataBit, addrReceived, lun0ready,
lun1ready, intCounter, addr3Block, addr2Page, addr1ColH, addr0ColL) =
  t1.tLunSelected!targRequest -> t1_setSR6.tLunSelected!false ->
  t1.tLunSelected!targRequest -> t1.tLunSelected!retrieveParameters ->
  (t1.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))

```

Figure A.6: SCXML vs generated CSP code for state T\_RPP\_ReadParams

## A.2 ONFI

The above CSP is considerably more compact than the original SCXML, and the reader will be forgiven for questioning the efficiency of the intermediate XML step.

The advantages should become clear when one considers the increase in size of the CSP code when a more realistic amount of state is included. The state variables for a target, as well as the extra state detailed in section 3.1.3, must be included in every reference to a process. Figure A.6 compares the resulting CSP code of the state T\_RPP\_READPARAMS with the SCXML from section 4.2.4. Clearly to write this kind of code by hand would be laborious, and more importantly, prone to data entry errors.

## Appendix B

# Correspondence with ONFi

### B.1 SR[6] update

In an email dated 3 July 2008, in response to the query we raised in relation to section 5.1.2, Michael Abraham of Micron suggested the following amendment to the specification.

I agree with your assessment for query #2. The behavioral flow has a problem and needs to be updated. Here's how I see that this should be handled:

1. T\_RPP\_ReadParams should remove "1. Request LUN tLunSelected clear SR[6] to zero."
2. T\_RPP\_ReadParams should remove "2. R/B# is cleared to zero."
3. L\_Idle\_RpPp should add a line that says, "1. lunStatus[6] is cleared to zero. lunStatus[6] value is indicated to the Target."

I think that these changes would automatically cause R/B# to be set LOW. RPP is a target-level command and no operations should be occurring on any of the other LUNs while this command is executing. I believe that is why we did a target-level update of R/B# in T\_RPP\_ReadParams.

I haven't looked at the other flows mentioned (Set/Get Features, Read Unique ID), but I'd think that the problem could be similarly resolved.



## Appendix C

# Technical architecture

The files `ONFI.csp` and `ONFI-mandatory.csp` draw together all the relevant code using CSP's `include` command. Figure C.1 shows their contents. Note that the referenced files are the same in both files, apart from the auto-generated CSP files for the host, target, and LUN state machines, which vary between the mandatory and full specification. To perform the checks one simply loads either file into the FDR2 refinement-checking program and double-clicks the refinement of interest.

`header.csp` contains declarations of datatypes and CSP channels, as described in section 3.1.2.

`SR6.csp` contains the *READYBUSY* process, as described in section 4.5.1.

`lun-innards.csp` contains the implementation of the *LUN\_INNARDS* process described in section 4.5.2.

`footer.csp` is the file that actually composes the separate CSP processes into a single system, and contains the refinements and other checks described in section 4.7.


<code>ONFI.csp</code>	<code>ONFI-mandatory.csp</code>
<code>include "header.csp"</code>	<code>include "header.csp"</code>
<code>include "host-software.csp"</code>	<code>include "host-software-mandatory.csp"</code>
<code>include "host-hardware.csp"</code>	<code>include "host-hardware-mandatory.csp"</code>
<code>include "target.csp"</code>	<code>include "target-mandatory.csp"</code>
<code>include "SR6.csp"</code>	<code>include "SR6.csp"</code>
<code>include "lun.csp"</code>	<code>include "lun-mandatory.csp"</code>
<code>include "lun-innards.csp"</code>	<code>include "lun-innards.csp"</code>
<code>include "footer.csp"</code>	<code>include "footer.csp"</code>

Figure C.1: `ONFI.csp` and `ONFI-mandatory.csp`

## Appendix D

# Optimizing CSP for FDR2

During development of the model, the boundaries of the FDR2 model-checker program were continually being tested.

Failure in FDR2 is invariably announced with the status message `'failed to compile ISM'` and the error icon: .

### D.1 State space

Reducing the state space is the primary method of addressing this problem. This is achieved in several ways:

- Eliminating unnecessary state variables. Is a higher level of abstraction, with fewer state variables, appropriate for the model in question?

In this project it was necessary to disregard the state of the actual flash array (i.e. the bits used for storing data). This has no effect on the sequences of events that the device exhibits, and hence the model remains valid for the purposes of verifying the specification.

- Resetting state variables to a known value on return to key states, so long as this does not affect the semantics of the model. This reduces the number of state-space possibilities of the key states.
- Setting bounds on state variables. For example, there may be a counter that is known never to go above a certain value. On incrementing the counter, modulo arithmetic or a simple maximum check can be used to set a bound on that counter.

In some cases it may be possible to replace an integer counter with a simple boolean 0 or 1.

## D.2 Stack limit

Correspondence with Formal Systems, the makers of FDR2, revealed that in certain cases the stack limit was being reached. This is easily resolved via the bash command:

```
ulimit -s 262144
```

This increases the stack space available to all commands subsequently executed by that shell. The command can also be added to the user's `bashrc` file to avoid the necessity of typing it each time.

## D.3 Operating system and architecture

Since FDR2 is available on both Linux and Solaris platforms, it is worth (as a last resort) attempting to compile the model on an alternative system. In this project the SPARC Solaris platform was found to have fewer compilation problems, though no systematic investigation was undertaken.

## Appendix E

# Event sequence example: Single 'Read' operation

As noted in section 4.1, initial work on the project included a manually-created spreadsheet showing the sequence of events during some common operations. Below is the sequence for a normal read operation, for a hardware-based host that does not require a Read Status operation.

Light blue rows indicate communication taking place between target and LUN, while red text indicates an action is taking place, perhaps internally.

Standard read from LUN 1								LUN behaviour										
	tbStatusOut	tbChgCol	tbCopyback	tLunSelected	tLastCmd	tReturnState	tbStatus78hReq	R/B#		lunStatus	lunFail[]	lunLastConfirm	lunReturnState	lunStatusCmd	lunStatusIly	lunInterleave	lunblvNextCmd	
<b>T Idle</b>	F	F	F	0 FFh	T_Idle	F		1		<b>L Idle</b>	01000000	0 FFh	L_Idle	70h	0h	F	??	
Command cycle received																		
<b>T Cmd Decode</b>	F	F	F	0 FFh	T_Idle	F		1										
Command 00h (Read) decoded																		
<b>T RD Execute</b>	F	F	F	0 FFh	T_Idle	F		1										
tbStatusOut set to FALSE																		
<b>T RD AddrWait</b>	F	F	F	000h	T_Idle	F		1										
Address cycle received																		
<b>T RD Addr</b>	F	F	F	000h	T_Idle	F		1	[store the address cycle received]									
More address cycles required																		
<b>T RD AddrWait</b>	F	F	F	000h	T_Idle	F		1										
Address cycle received																		
<b>T RD Addr</b>	F	F	F	000h	T_Idle	F		1	[store the address cycle received]									
More address cycles required																		
<b>T RD AddrWait</b>	F	F	F	000h	T_Idle	F		1										
Address cycle received																		
<b>T RD Addr</b>	F	F	F	000h	T_Idle	F		1	[store the address cycle received]									
More address cycles required																		
<b>T RD AddrWait</b>	F	F	F	000h	T_Idle	F		1										
Address cycle received																		
<b>T RD Addr</b>	F	F	F	000h	T_Idle	F		1	[store the address cycle received]									
More address cycles required																		
<b>T RD AddrWait</b>	F	F	F	000h	T_Idle	F		1										
Address cycle received																		
<b>T RD Addr</b>	F	F	F	000h	T_Idle	F		1	[store the address cycle received]									
All address cycles received																		
<b>T RD LUN Execute</b>	F	F	F	100h	T_Idle	F		1	Issues read Page to tLunSelected, requests all idle LUNs not selected to turn off their output buffers	Target request received								
Unconditional																		
<b>T RD LUN Confirm</b>	F	F	F	100h	T_Idle	F		1		<b>L Idle TargetRequest</b>	01000000	0 FFh	L_Idle	70h	0h	F	??	
Command cycle of 30h received																		
<b>T RD Cmd Pass</b>	F	F	F	100h	T_Idle	F		1	Pass command received to LUN 1	<b>L RD WaitForCmd</b>	01000000	0 FFh	L_Idle	70h	0h	F	??	
Command Cycle 30h received																		

