Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B *

Kriangsak Damchoom and Michael Butler

University of Southampton
United Kingdom
{kd06r,mjb}@ecs.soton.ac.uk

Abstract. Event-B is a formal method used for specifying and reasoning about systems. Rodin is a toolset for developing system models in Event-B. Our experiment which is outlined in this paper is aimed at applying Event-B and Rodin to a flash-based filestore. Refinement is a useful mechanism that allows developers to sharpen models step by step. Two uses of refinement, feature augmentation and structural refinement, were employed in our development. Event decomposition and machine decomposition are structural refinement techniques on which we focus in this work. We present an outline of a verified refinement chain for the flash filestore. We also outline evidence of the applicability of the method and tool together with some guidelines.

Key words: refinement, event decomposition, machine decomposition, file system, flash memory, proof, Event-B, Rodin

1 Introduction

Hoare and Misra [14] outline the importance of undertaking experiments involving the application of theories and tools in order to push forward scientific progress in formal methods. Experiments help us to understand the strengths and weaknesses of theories and tools. A flash-based filestore has been selected as case study for our experiment. This case study was proposed as a challenging system by Joshi and Holzmann [19]. As stated in [19], the challenge is how to deal with accidental failures that may occur while performing operations on a flash memory. For example, how do we cope with power loss or sudden reboot? How do we manage the data consistency when flash instructions being performed fail? The flash architecture we chose is the ONFI (Open NAND Flash Interface) specification proposed in [16]. This specification is open and is commonly referenced by researchers who are working in this area.

A flash array has physical characteristics that constrain the way it is used. Taking account of physical characteristics and failure management is required.

* This work was part of the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

Reading and writing of files to the flash array are expected to be consistent with an abstract model of a filesystem.

Our experiment presented in this paper is the development of a verified refinement chain for a flash-based filestore using Event-B and the Rodin platform. This experiment is an extension of the work we presented in [11] where we outlined a model of a tree-structured file system. The extension we address here consists of replacing the abstract file system by the flash specification and dealing with fault-tolerance. In Section 12 we discuss some related work on applying formal methods to the file store problem. A distinguishing feature of our treatment of the file store problem is the use of multiple levels of refinement to relate an abstract model, with large atomic reads and writes on abstract data structures, to a model with more complex concrete data structures and more fine-grained atomic steps. The use of multiple levels of refinement means that the abstraction gap is relatively small at each stage which means the gluing invariants required for refinement verification are relatively simple. Simpler gluing invariants are easier for modellers to formulate and lead to simpler proof obligations. We believe the relative ease of the proof effort, reported in Section 11, testifies to this. Another distinguishing feature of our development is the use of machine decomposition to partition the development after several refinement steps. The partitioning led to sub-models that were refined separately. While it is well-known that decomposition is critical for scaling of formal development, it is rare to find examples of its application in practice. Our file store development represents an exemplar of multi-level refinement and of machine decomposition that we believe others could learn from. This role as an exemplar is the main contribution of the paper.

Two uses of refinement were employed in our development: horizontal and vertical refinements (details are given in Section 3). The horizontal development was mainly presented in [11]. (In this paper, we focus on the vertical refinement.) We first used horizontal refinement in an incremental way to construct the file system model. The model started with an abstract tree structure. After that, new features were gradually added in each refinement step. We finally got five layers of specification describing an abstract file system.

Vertical refinement was later used to introduce more design details in order to map the abstract file system to the flash architecture. In the case of vertical refinement, while refining the file system down to the flash specification, the event-decomposition technique [6] was used to decompose events like *readfile* (read the whole content of the given file from the storage into the memory buffer) and *writefile* (write the whole content of the given file on the buffer to the storage) into three sub-events, *start*, *step* (read or write a page) and *end*.

We also applied the machine decomposition technique [6] to decompose the machine of the last refinement into two sub-machines representing the specification of the file system layer and the flash interface layer. The reason we do this is to explore further refinements of the flash model separately from the file system model.

The paper begins with an introduction to Event-B and Rodin in Section 2. The refinement and event-decomposition techniques used in our development

are outlined in Section 3 and 4. An overview of an abstract file system is given in Section 5. Vertical refinement and event-decomposition used to link the file system to the flash specification are discussed in Section 6–8. Machine decomposition and further refinement focusing on the flash specification are outlined in Section 9 and 10. Proof statistics, related work and conclusion are discussed in Section 11, 12 and 13, respectively.

Note: An archive of our development in Rodin may be downloaded[1]. This can be imported by the Rodin tool release 0.9.2.1 or later[2].

2 Event-B and Rodin

Event-B [3] is an extension of the B-method [1] for specifying and reasoning about systems. An Event-B model is described in terms of contexts and machines. Contexts [4, 5] contain the static parts of a model. Each context may consist of carrier sets and constants as well as axioms that are used to describe the properties of those sets and constants. Contexts may contain theorems which are required to follow from the preceding axioms and theorems. Machines [4, 5] represent the dynamic part of an Event-B model consisting of variables and actions. A machine is made of a state, which is defined by means of variables, invariants, events and theorems. The theorems of a machine are required to follow from the context and the invariants of that machine. Variables are typed as mathematical objects such as sets, relations, numbers, etc. Variables are constrained by invariants. Invariants are expected to be preserved whenever variable values change. This must be proved through the discharge of proof obligations [4].

A machine contains a number of atomic events which model the way that a system may evolve. In general, an event is composed of four elements: name, parameter, guard and action. The guard is the necessary condition for the event. The action determines the way in which the state variables are going to evolve when performing the event [4]. An event is guarded and atomic and may be performed only when its guard holds. When the guards of several events hold at the same time, then only one event may be performed at that time. The event to be performed is non-deterministically chosen.

Refinement is the main development method supported by Event-B. In Event-B, an event of an abstract machine may be refined by several corresponding events in a refined machine. A refined machine may also have additional events that are refinements of *skip* rather than being refinements of abstract events. Note this is more flexible than the usual approach in, for example, Z, VDM or "classical" B, where there is a strong one-to-one correspondence between abstract and concrete events.

Rodin [4] is an open and extensible toolset for specifying and verifying system models in Event-B. It contains a database of modelling elements used for constructing system models such as variables, invariants, events, etc. The Rodin

[1] http://deploy-eprints.ecs.soton.ac.uk/125/
[2] www.event-b.org

toolset is accompanied by various useful plug-ins such as a proof-obligation generator, provers, model-checkers, UML transformers, etc [9].

3 Refinement Strategy

Incremental refinement has been used as our strategy to develop a model of a flash-based file system. Two uses of refinement were employed in our development: *feature augmentation* (or horizontal refinement) and *structural refinement* (or vertical refinement) [8]. Feature augmentation is aimed at introducing new requirements or properties which are not addressed in the initial model or may be postponed to other levels. Thus, in each refinement step, additional state variables and related events might be added to incorporate those features which are introduced. The system models will be enlarged gradually when new properties are introduced. The purpose of structural refinement, on the other hand, is to replace an abstract structure with more design details in each refinement step down to an implementation. This kind of refinement may involve data refinement, event decomposition and machine decomposition.

In the development presented in [11], feature augmentation was used in an incremental way to develop a model of a flash-based file system. That is, we began with a small set of features and then enlarged the model by introducing new features in each refinement step. We finally got five levels of a specification describing an abstract file system. That is, the specification is the abstract model plus a series of feature augmentations. As stated in [11], we regard the full chain of augmentation (horizontal) refinements as constituting *the specification*, not just the most abstract level.

Structural refinement, which is the focus of this paper, was used to relate the abstract file system with the specification of the flash interface layer. This kind of refinement was used to decompose the events *readfile* and *writefile* into sub-events in order to map them with *page-read* and *page-program* interfaces provided by the flash interface layer. Details will be given in Section 6 and 7.

4 Event Decomposition in Event-B

While refining a model we may find that some (atomic) events can be split into sub-events. We can decompose this kind of event through a refinement step. Among those sub-events which are split, at least one event refines the abstract event, while other sub-events refine *skip*. In our case, for example, instead of writing the whole content in one step, the abstract file write can be partitioned into sub-events: (i) start write (set an initial state and buffers), (ii) write a single page (occurs once for each page of a file) and (iii) end write (reset the state and buffers of the given file). Note that we achieve this form of event decomposition using the standard refinement rules of Event-B which allows for the introduction of events that refine *skip* in a refinement [3].

To understand more about event decomposition, event refinement diagrams proposed in [6] will be used to explain how an atomic event can be decomposed

into sub-events. Fig. 1 shows an example of such a diagram. In the figure, the root represents an abstract event which is partitioned into events *start*, *step*, and *end* in a refinement. A solid line indicates that the *end* event refines the *abs_evt* event. That means the *end* event will be proved to refine the abstraction. The dashed lines state that both *start* and *step* refine *skip*. The oval represents a quantifier that specifies multiple interleaved instances of an event (i will range over some set). Order, from left to right, constrains the order in which events have performed. A *step*(i) event can be performed only when the *start* event is completed, and *end* can be performed only when all *step*(i) events have been occurred. The order amongst the *step*(i) events is nondeterministic. In Event-

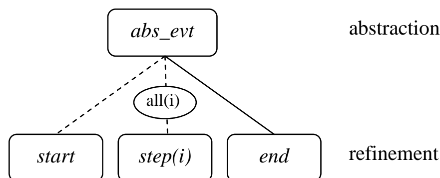


Fig. 1. An example of event refinement diagram

B, there are no explicit sequencing operations. Events are non-deterministically performed when their guards hold. Thus, in order to control the order of event execution, each event must be guarded by using additional states or flag variables. For example, in order to start writing a single page, the given file must be in the writing state. Thus, a *writing* state should be introduced and used to construct guards of events that we want to control.

The event refinement diagrams are used as an aid to constructing and understanding the formal models rather than being formal objects themselves. As outlined in [6] , the diagrams were inspired by Jackson Structured Design (JSD) diagrams [18]. In the future, we plan to investigate a more formal incorporation of event refinement diagrams into the refinement proof obligations.

5 Outline of Abstract File System

In the development presented in [11], feature augmentation was used in an incremental way to develop the model of an abstract file system. That is, we began with a small set of features and then augmented the model by adding new features in refinement steps. Additional state variables and events which are related to the new features were introduced in each step. The event-extension feature[3] provided by the tool was mainly used to develop this refinement chain. In each refinement step, when new features were introduced the related events were extended by adding more details or constraints corresponding to those features. The event extension may involve adding new parameters, guards and actions.

[3] Event extension is a new feature of Rodin.

The layered specification of the abstract file system is briefly described as follows.

Abstraction. Tree properties and basic operations affecting the tree structure (*create*, *delete*, *move* and *copy*) were firstly specified in this level. No-loop and reachability (all objects in a tree are reachable from the *root*) are two main properties which were the focus of verification effort.

First refinement. Files and directories were introduced. In the abstraction, files and directories are treated in the same way as *objects* which are nodes of the tree structure. In this level, *objects* was replaced by *files* and *directories*. That means an object can be either a file or a directory. The abstract event *create* was refined into *crtfile* (create a file) and *mkdir* (make a directory).

Second refinement. File content was introduced in this level. Additional constraints and events related to file content are also addressed. For example, each file has a content, an existing file must be opened before reading or writing.

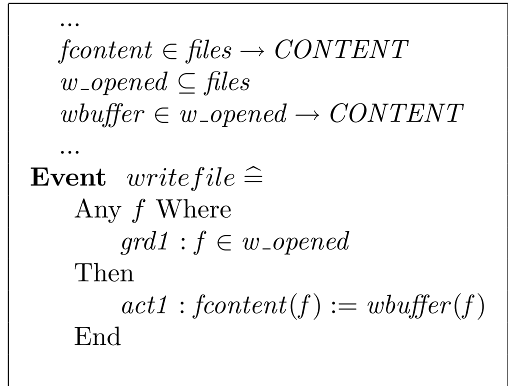
Third refinement. Access permissions and related constraints were introduced. For instance, each object has an owner, a group-owner and a list of permissions. The user who issues read- or write-request must have the right to read or write on the given file.

Fourth refinement. Additional properties which were not addressed in [11] – such as objects' name, creation date and last modification date – were introduced here.

Fig. 2 shows the definitions of three variables of the abstract specification along with an abstract file write event, named *writefile*, of the abstract file system. The *writefile* event writes the whole content of the given file f from the write buffer (*wbuffer*) into the storage in one step. Here *fcontent* represents the content of each file on the storage, *w_opened* is a set of files which are opened for writing, and *CONTENT* is defined as a sequence of *DATA* in a context seen by this abstract machine.

6 Vertical Refinement

The purpose of this section is to outline the decomposition of the abstract events *readfile* and *writefile*. The decomposition is based on the assumption that the content of the file is read from or written to the storage one page at a time. As shown in Fig. 3 (b), for example, instead of writing the buffer content into the storage in one step, we introduced an intermediate variable named *fcont_tmp*. This variable behaves like a shadow disk used for accumulating the content of the pages as they are written one at a time. This shadow becomes the actual content of that file only when all pages have been written to the shadow. The use of this shadow allows us to deal with faults that may occur during writing a file – if a fault occurs, we discard the shadow and keep the original. The use of the shadow is an abstraction of the fact that when writing a file at the implementation level



...
$fcontent \in files \rightarrow CONTENT$
$w_opened \subseteq files$
$wbuffer \in w_opened \rightarrow CONTENT$
...
Event $writefile \mathrel{\widehat{=}}$
Any f Where
$grd1 : f \in w_opened$
Then
$act1 : fcontent(f) := wbuffer(f)$
End

Fig. 2. Event *writefile* of the abstract file system

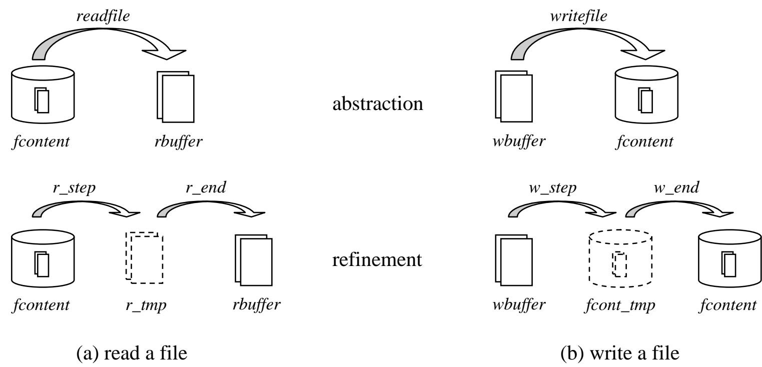


Fig. 3. A diagram of refining events *readfile* and *writefile*

we use fresh pages on the flash array rather over-writing the pages used for the previous version of the file. Additional details are explained in Section 7.

Note: Because of space constraints, instead of detailing the decomposition of both file read and file write which are similar, we will present only file write which is more interesting. Full details of the specification can be found in the archive mentioned in Section 1.

7 Decomposing Event *writefile*

Fig. 4 (a) shows an event refinement diagram for the *writefile* event which is decomposed into three sub-events: *w_start* (start write), *w_step* (write one page at a time) and *w_end* (end write, when all pages are written completely). Event *w_end* refines *writefile* of the abstraction while *w_start* and *w_step* refine *skip*. This diagram states that *w_start* must be performed before *w_step*. Event *w_step* will be repeated until all pages are written or programmed into the flash device.

In case of failures (see Fig. 4 (b)), in the abstraction, the *writefileFail* event does nothing (i.e. *skip*). The content of file on the storage is not changed and all memory buffers are released.

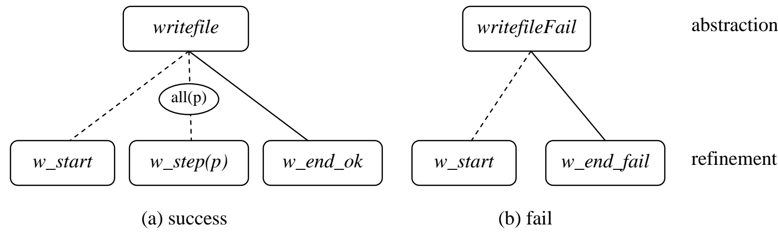


Fig. 4. Refinement diagram of event *writefile*

Fig. 5 shows machine invariants in this refinement step. Variable *fcont_tmp* represents temporary content of the file while it is in the writing state. This variable behaves like a shadow content of the file being written, as already discussed. This shadow content becomes an actual content (*fcontent*) when all pages have been written to the shadow. No change is made to *fcontent* if writing a file fails at any point from the start to the end of writing a file. That means the content of that file will be same as the previous state. We specified *writing* as a set of opened files which are in the writing state. Variable *wbuffer* represents a write-buffer of each writing file. Invariant $inv6.3$ states that for any file f which is in the writing state, the temporary contents of f will be a subset or equal to the content on its writing buffer.

Fig. 6 shows the refinement of event *writefile* when it is split into *w_start*, *w_step* and *w_end* (in cases of *success* and *fail*). Consider the *w_start* event. In order to start writing a file, the given file must be opened for writing and not

$$
\begin{aligned}
&inv6.1 : writing \subseteq w_opened \\
&inv6.2 : fcont_tmp \in writing \rightarrow CONTENT \\
&inv6.3 : \forall f \cdot f \in writing \Rightarrow fcont_tmp(f) \subseteq wbuffer(f)
\end{aligned}
$$

Fig. 5. Machine invariants of the refinement

already in the writing state (see *grd1* and *grd2* of event *w_start*). Event *w_step* writes the contents of page i from the write buffer (*wbuffer*) into *fcont_tmp*. In order to do this the given file must be in the writing state (see *grd1*). The page being written must be a page in the write buffer that has not already been written to the storage (see guards *grd4* and *grd5* of event *w_step*). Event *w_end_ok* is enabled when all pages have been written (*grd2*) and the file is in the writing state. The effect of *w_end_ok* is to overwrite the existing file content with the shadow content.

Guard *grd2* of the *w_end_ok* event and Invariant *inv6.3* play an important role in proving that the *w_end_ok* event is a correct refinement of the *writefile* event (given in Fig 2). Namely, the gluing invariant, *inv6.3*, is used to show that *fcont_tmp(f)* is equal to *wbuffer(f)* when the guards of the *w_end_ok* event holds.

8 Linking the Abstract File System to the Flash Interface Layer

This section outlines our model of the flash specification, which is based on the ONFI specification given in [16], and shows how it is related to the abstract file system via data refinement. We first describe an abstract specification of the flash in Section 8.1 and then show the refinement of the file system layer when the flash specification is included.

8.1 Abstract Flash Interfaces Layer

An ONFI-based flash device is a collection of LUNs (Logical Units). Each LUN is composed of a number of blocks. Each block has a number of pages. Each page is a sequence of data items.

Flash pages are accessed via row addresses consisting of a LUN, a block number within a LUN and a page number within a block. A flash device can be specified as an array of pages which are identified by row addresses:

$$flash \in RowAddr \rightarrow PDATA$$

where *RowAddr* is a set of possible row addresses. *PDATA* represents a page data within each page. To realise the file system layer, we assume that each *PDATA* is composed of file data, the object identity to which the data belongs, the logical

Event $w_start \mathrel{\widehat{=}}$
Any f Where
$grd1 : f \in w_opened$
$grd2 : f \notin writing$
Then
$act1 : writing := writing \cup \{f\}$
$act2 : fcont_tmp(f) := \varnothing$
End

Event $w_step \mathrel{\widehat{=}}$
Any f, i, cnt Where
$grd1 : f \in writing$
$grd2 : i \in \mathbb{N}$
$grd3 : cnt \in DATA$
$grd4 : i \mapsto cnt \in wbuffer(f)$
$grd5 : i \notin dom(fcont_tmp(f))$
Then
$act1 : fcont_tmp(f) := fcont_tmp(f) \cup \{i \mapsto cnt\}$
End

Event w_end_ok refines $writefile \mathrel{\widehat{=}}$
Any f Where
$grd1 : f \in writing$
$grd2 : dom(wbuffer(f)) = dom(fcont_tmp(f))$
Then
$act1 : fcontent(f) := fcont_tmp(f)$
$act2 : writing := writing \setminus \{f\}$
$act3 : fcont_tmp := \{f\} \lhd fcont_tmp$
End

Event $w_end_fail \mathrel{\widehat{=}}$
Any f Where
$grd1 : f \in writing$
Then
$act1 : writing := writing \setminus \{f\}$
$act2 : fcont_tmp := \{f\} \lhd fcont_tmp$
End

Fig. 6. Decomposition of the *writefile* event

page-id (or page index in the view of file system) and a version number. Fig. 7 represents the structure of *PDATA*. We model each component of *PDATA* by a projection function. For example, the file data stored in a *PDATA* is modelled by *dataOfpage* (*axm1*). The other projections represent file object, page index and version number. A set of version numbers (*VERNUM*) is used to record the version of data which is programmed in each page.

$$\begin{aligned}
&axm1 : dataOfpage \in PDATA \rightarrow DATA\\
&axm2 : objOfpage \in PDATA \rightarrow OBJECT\\
&axm3 : pidxOfpage \in PDATA \rightarrow \mathbb{N}\\
&axm4 : verOfpage \in PDATA \rightarrow VERNUM
\end{aligned}$$

Fig. 7. A structure of PDATA

The flash interface layer provides two main interfaces to the file system layer. The first is *page_read*, read a page of data from a given row address, and the second is *page_program* (or *page_write*), write a page of data into the flash device at a given row address. These two interfaces will become part of the events *r_step* and *w_step* of the file system layer.

8.2 Relating the File System Layer with the Flash Interface Layer

In this refinement step, flash properties are introduced together with variables used to relate those two layers. Variables *fcontent* and *fcont_tmp* of the file system layer are replaced by *fat* and *fat_tmp* respectively. The variable *fat* represents the table of contents of each file. This table is a mapping of each file to a table that maps each logical page-id of the file to its corresponding row address in the flash. The corresponding row address represents the location (in the flash) in which the content of that page is stored.

The properties mentioned above are described by the invariants given in Fig. 8. Invariants *inv7.3* and *inv7.4* are gluing invariants introduced to relate the abstract variables *fcontent* and *fcont_tmp* with the concrete variables *fat* and *fat_tmp* respectively. They play an important role in proving the correctness of this refinement. Variable *programmed_pages* represents the row addresses of pages that have already been programmed or written, while *obsolete_pages* is a set of programmed pages that are invalid to be used. Invariants *inv7.8* and *inv7.9* were introduced to relate the content of file with the actual content on the flash device. For instance, *inv*7.8 says that for any page whose version equals to the current version of the file to which the page belongs, the data of that page will be the data of the given page-id of that file.

Fig. 9 illustrates how the file write of the abstract file system is replaced by the flash specification. The top diagram represents the abstract file write

$$
\begin{array}{l}
inv7.1 : fat \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr)\\
inv7.2 : fat_tmp \in writing \rightarrow (\mathbb{N} \nrightarrow RowAddr)\\
inv7.3 : \forall f \cdot f \in files \Rightarrow dom(fat(f)) = dom(fcontent(f))\\
inv7.4 : \forall f \cdot f \in files \wedge f \in writing \Rightarrow dom(fat_tmp(f)) = dom(fcont_tmp(f))\\
\\
inv7.5 : flash \in RowAddr \rightarrow PDATA\\
inv7.6 : programmed_pages \subseteq RowAddr\\
inv7.7 : obsolete_pages \subseteq programmed_pages\\
\\
inv7.8 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in files\\
\quad \wedge\ verOfpage(p) = curr_version(objOfpage(p)) \wedge pidxOfpage(p) \neq 0\\
\quad \Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcontent(objOfpage(p))\\
\\
inv7.9 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in writing\\
\quad \wedge\ verOfpage(p) = writing_version(objOfpage(p)) \wedge pidxOfpage(p) \neq 0\\
\quad \Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcont_tmp(objOfpage(p))\\
\ldots
\end{array}
$$

Fig. 8. Machine invariants of replacing the file system by the flash specification

which is composed of three sub-events: *w_start*, *w_step* and *w_end*. The bottom diagram represents the refinement where *w_step* is refined by event *pagewrite*. In this event, *page_program* will be called in order to write the content of each page into the flash device. When each page has been programmed successfully, the *fat_tmp* will be updated. Finally, the *fat_tmp* will be copied to *fat* when all pages have been completely programmed into the flash device.

Fig. 10 shows the *pagewrite* event which is a refinement of the *w_step* event. The *pagewrite* event will look for an available page on the flash (*grd6-grd7*) in order to write the content of page number i on the *wbuffer*. Parameter r represents a row address within the flash. Guards *grd9-grd12* describe the contents of *pdata* to be written to the flash. Action *act1* updates the temporary *fat* table of the file *f*. Action *act2* sets the content of the flash at row number r equal to *pdata*. The row address identifying that page will be set as a programmed page by *act3*.

9 Machine Decomposition

The aim of this section is to decompose the machine into a file system machine, modelling the file system layer, and a flash machine, modelling the flash interface layer. As a result, further refinements of the flash interface layer can be explored separately. The machine decomposition we apply here follows the style described in [6]. Namely, machine variables and events are partitioned into sub-machines. Sub-machines interact with each other via synchronisation over shared parameterised events.

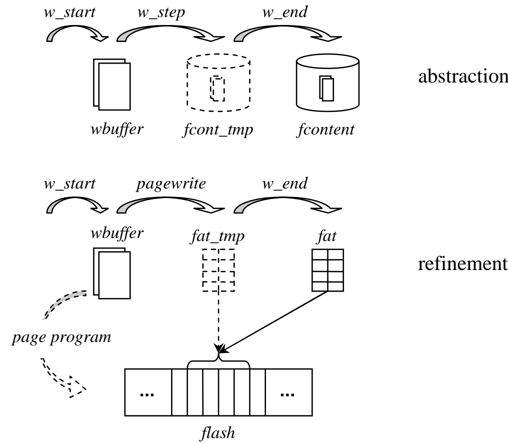


Fig. 9. A diagram of mapping *writefile* to the flash specification

Event $pagewrite$ refines $w_step \mathrel{\widehat{=}}$

Any $f, i, cnt, r, pdata$ Where

$grd1 : f \in writing$

$grd2 : i \in \mathbb{N}$

$grd3 : cnt \in DATA$

$grd4 : i \mapsto cnt \in wbuffer(f)$

$grd5 : i \notin dom(fat_tmp(f))$

$grd6 : r \in RowAddr$

$grd7 : r \notin programmed_pages$

$grd8 : pdata \in PDATA$

$grd9 : verOfpage(pdata) = writing_version(f)$

$grd10 : objOfpage(pdata) = f$

$grd11 : lpidOfpage(pdata) = i$

$grd12 : dataOfpage(pdata) = cnt$

Then

$act1 : fat_tmp(f) := fat_tmp(f) \cup \{i \mapsto r\}$

$act2 : flash(r) := pdata$

$act3 : programmed_pages := programmed_pages \cup \{r\}$

End

Fig. 10. The refinement of the w_step event

Fig. 11 shows a diagram of machine decomposition illustrating the decomposition of the events *pagewrite* and *pageread*. The top layer represents the file system sub-machine consisting of machine variables *fat*, *fat_tmp*, *wbuffer*, and so on. The lower layer represents the flash interface sub-machine which contains machine variables named *flash*, *programmed_pages* and *obsolete_pages*. The ovals represent synchronisation over shared parameterised events between the sub-machines. In this case, both sub-machines interact by synchronising over the *page_write* and the *page_read* events.

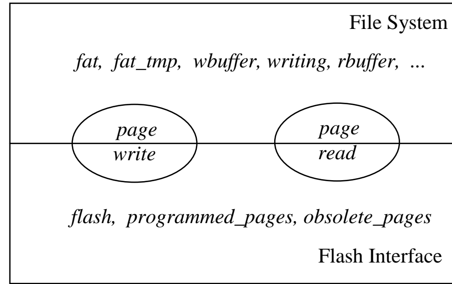
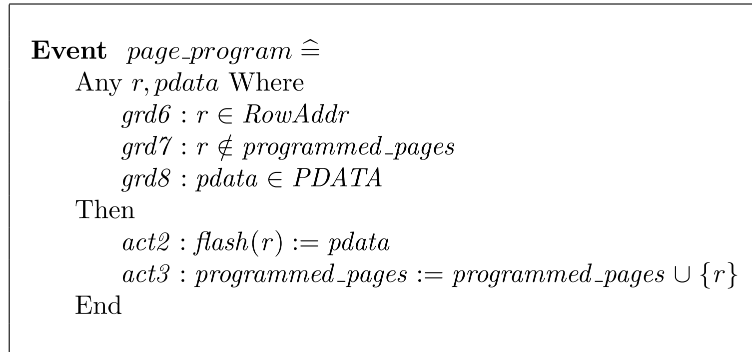


Fig. 11. A machine-decomposition diagram focusing on events *page_read* and *page_write*

At this point, for example, we partitioned the *pagewrite* event given in Fig. 10 following the approach of [6] and got a specification of the *page_program* event of the flash interface layer which is shown in Fig. 12. We also got a specification of the *pagewrite* event of the file system layer given in Fig 13. Parameters r and *pdata* represent shared parameters which are used for an interaction between these two events.



Event *page_program* $\widehat{=}$
Any $r, pdata$ Where
$grd6 : r \in RowAddr$
$grd7 : r \notin programmed_pages$
$grd8 : pdata \in PDATA$
Then
$act2 : flash(r) := pdata$
$act3 : programmed_pages := programmed_pages \cup \{r\}$
End

Fig. 12. An abstract *page_program* of the flash interface layer

Event $pagewrite \mathrel{\widehat{=}}$
Any $f, i, cnt, r, pdata$ Where
$grd1 : f \in writing$
$grd2 : i \in \mathbb{N}$
$grd3 : cnt \in DATA$
$grd4 : i \mapsto cnt \in wbuffer(f)$
$grd5 : i \notin dom(fat_tmp(f))$
$grd6 : r \in RowAddr$
$grd8 : pdata \in PDATA$
$grd9 : verOfpage(pdata) = writing_version(f)$
$grd10 : objOfpage(pdata) = f$
$grd11 : lpidOfpage(pdata) = i$
$grd12 : dataOfpage(pdata) = cnt$
Then
$act1 : fat_tmp(f) := fat_tmp(f) \cup \{i \mapsto r\}$
End

Fig. 13. Event *pagewrite* of the file system layer

After decomposition we finally got a machine specifying the flash interface layer which consists of events *page_program* and *page_read* that can later be refined separately from the specification of the file system. We also got a machine specifying the file system with *pagewrite* and *pageread* plus the other events from earlier refinement such as *w_start* and *w_end*.

10 Further Refinements

Further refinements are focused on the flash interface layer. After decomposition, the flash model is refined separately by adding more details of the flash specification. For example, each LUN has at least one page register used for buffering data. Writing a page is done in two phases. The first is writing the given data into a page register within the selected LUN and the second is programming the data on the page register into the flash array at the given row address. Similarly for reading page data, the data will be first transferred to the page register before it is read off chip into the memory buffer.

An additional event that we specify is *block-erase*. This event has the effect of erasing a given block in order to be reused for writing. The number of erasures per block is limited (the number is dependent on its manufacturing). The block which fails to erase will become a bad block which is no longer to be used. In order to reclaim a dirty block, the block should contain obsolete data and may have one or more pages whose data is still valid. All valid pages within the block being reclaimed must be relocated (moved to another fresh block). After all valid

pages have been relocated, the given block becomes obsolete and ready to be erased. That means only obsolete blocks are allowed to be erased.

These further refinements mentioned are the refinements of the flash interface layer which are refined separately from the model of the file system layer.

Note that the *wear-levelling* process[4] is an important feature that has not been covered in our development yet. It is in our on-going work.

11 Proofs

The proof statistics, given in Table 1[5], show that 540 proof obligations (POs) were generated by the Rodin tool. 501 POs (or 93%) were proved automatically while others were discharged by interactive proof. In the case of interactive proofs, almost 80% of proof steps involved instantiation of universal quantifiers while the rest involved adding hypotheses, case distinctions, etc. In this table, MCH0 represents an abstract model; MCH1 up to MCH4 represent the first up to the fourth horizontal refinements, MCH5 – MCH7 represent the vertical refinements. MCH_FL0 up to MCH_FL3 represent an abstract model (MCH_FL0) of the flash interface layer and its refinements.

Table 1. Proof statistics

Machines	Total POs	Automatic	Interactive
MCH0	35	22	13
MCH1	57	49	8
MCH2	33	32	1
MCH3	37	34	3
MCH4	26	26	0
MCH5	27	26	1
MCH6	31	30	1
MCH7	109	97	12
MCH_FL0	8	8	0
MCH_FL1	110	110	0
MCH_FL2	57	57	0
MCH_FL3	9	9	0
Overall	540	501 (93%)	39 (7%)

In each step of iteration of modelling, modification and proof, POs generated by the tool were used as guidelines for modelling and reasoning about the model. For example, they were used to determine which gluing invariant should be added

[4] A technique used for prolonging the life time of flash memory covering reclaiming and erasing blocks within a flash chip.

[5] These proof statistics are slightly different from the table given in [11] because we have introduced events for deleting a file and removing an empty directory in MCH1.

to the machine (e.g. $inv6.3$ given in Fig. 5), which guard should be added to the event in order to strengthen the model, as well as which form of expressions should be specified to make prove easier. For instance, specifying an expression like $pg \mapsto obj \in objOfpage$ is easier to discharge than $objOfpage(pg) = obj$. As a result, this technique means we get a higher degree of automatic proof.

Results for automatic proof are good, but there is room for improvement. In principle, when any change is made, Rodin has the ability to avoid re-running proofs that are still valid. However, in some cases, some (unnecessary) proofs need to be re-run when some changes are made. As a result, if there is a large number of POs to be reproved and it can take a lot of time to re-run unnecessary proofs whenever the model changes.

12 Related work

A number of formalisations of a file system have been developed by other researchers. For example, a specification of a visual file system in Z by Hughes [15] is focussed on tree properties and basic file operations affecting the tree structure, but file content and a manipulation of file content were not specified. The commonly referenced model developed by Morgan and Sufrin presented in [21] is a specification of a Unix file system in Z. In this specification, instead of using a tree structure, the location of each object is formulated as a sequence of directory names, which is the path of each object. This work is focused on file contents and naming operations used for manipulating these rather than structure manipulation operations such as directory copy and move. Based on the specification of Morgan and Sufrin, Freitas, Woodcock and Fu [13] have developed a verified model of the POSIX filestore accompanied with a representation and proof using the Z/Eves proof system. Since the filestore challenge was proposed by Joshi and Holzmann [19] in 2005, other researchers have addressed this challenge. For example, Butterfield, Freitas and Woodcock [10] have developed an abstract specification in Z of the ONFI specification [16]. In addition, Ferreira et al. [12] have developed and verified a specification of the Intel Flash File System Core [17] in VDM. HOL and Alloy were used as a theorem prover and model checker, respectively. Other work developed by Kung and Jackson [20] is a formal specification and analysis of a flash-based filestore in Alloy. [20] focusses on basic operations of a file system, such as *read* and *write*, and addresses fault tolerance and *wear-levelling* process.

The approach of refining *skip* events to achieve decomposition of the atomicity of events was used by Woodcock and Davies [22] to refine a file write operation with the Z notation. Like us, they use a shadow disk in the refinement. They show how the decomposition of the file write can be cast as either a forwards simulation or a backwards simulation. In our case, we work only with forward simulation as Rodin only supports forward simulation. We have not found this restriction in Rodin to cause any difficulties.

Other researchers mentioned above also report statistics from mechanical proof efforts. However, we found it difficult to perform a like-for-like comparison

of our results with others. Any comparison would depend heavily on the nature of the proof obligations and on the proof support provided in the language. For example, in Rodin, refined events may contain 'witness' clauses that are used to instantiate existential proof obligations. Without this, we would have a lot more interactive proofs whose only interactive step would be the provision of witnesses for existential quantifiers. By providing the witnesses directly in the model, we achieve a higher degree of automatic proof and the proofs are more robust against model changes. In the future it would be useful for researchers in this area to attempt to develop a common framework for comparing proof effort.

13 Conclusion and Discussion

We have presented a model of a flash-based filestore which was developed by using Event-B and Rodin. In this development, we have outlined the use of refinement for two different purposes. First refinement was used in feature augmentation (or horizontal refinement) and second for structural refinement (or vertical refinement).

Feature augmentation is a mechanism used for constructing a model of an abstract file system which was presented in [11]. Instead of specifying everything in one level that may give rise of proof difficulty, we decided to split the whole system features into sub-features. These sub-features were chosen to be introduced in refinement steps. We have found that this approach makes the model easier to construct and prove. In addition, we have found that the event-extension feature provided by the Rodin tool (release 0.9.x) makes models easier to refine. Namely, some changes can be made to the abstract levels individually and are propagated down automatically. This is in contrast to when we were developing the model of [11] using the Rodin tool release 0.8.2 that has no support for event-extension.

Structural refinement was used to relate the abstract file system with the flash specification. Event-decomposition is a structural refinement on which we focused in this paper. We have shown how the decomposition technique can be applied to our case study. This technique was used to partition atomic events *readfile* and *writefile* into a number of sub-events as explained in Section 7.

We have found that the event-decomposition technique is very effective for breaking an atomic event. It can be applied to other work whose events may require to be decomposed in order to cope with fault-tolerance or concurrency. An atomic event can be partitioned into sub-events that can be performed in an interleaved fashion. In addition, as can be seen in Section 7, we could deal with file write that may fail at any point between the start and the end of writing a file.

Additionally, in Section 9, the machine decomposition was employed to separate part of the flash interface layer from the file system layer. The purpose is to deal with further refinements of the flash interface layer separately. Those two layers interact with each other via the shared parameterised events. Based on this evidence, we believe that machine decomposition is useful for other de-

velopments whose specification involves a number of sub-systems that can be partitioned and refined separately. Rodin does not provide any tool to decompose machines directly, we still need to decompose machines manually using the editor of the Rodin tool. Thus, in the future, it would be useful if a machine-decomposition tool could be developed.

Although the proof statistics show a high degree of automatic proof, some improvements are still required. As explained in Section 11, in some cases, proofs are required to be re-run every time the model changes although they have already been proved. This is because Rodin currently uses a mixture of new and legacy provers and, while the new provers maintain sufficient information about used hypotheses to be able to avoid re-running proofs, the legacy provers do not. This is an engineering issue that is being addressed in Rodin.

As mentioned in the introduction, our file store development represents an exemplar of multi-level refinement and of machine decomposition that we believe others could learn from.

References

1. Abrial, J.-R.: The B Book. Cambridge University Press (1996)
2. Abrial, J.-R.: A system development process with Event-B and the Rodin platform. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) In ICFEM 2007. LNCS, vol. 4789, pp. 1–3. Springer (2007)
3. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. To be published by Cambridge University Press (2009)
4. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260. Springer (2006)
5. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamentae Infomatica. 1001–1026 (2006)
6. Butler, M.: Decomposition structures for Event-B. In: Integrated Formal Methods, iFM2009. LNSC, vol. 5423, pp. 20–38. Spinger (2009)
7. Butler, M., Yadav, D.: An Incremental development of the Mondex system in Event-B. Formal Aspects of Computing 20(1):61–77 (2008)
8. Butler, M., Abrial, J.-R., Damchoom, K., Edmunds, A.: Applying Event-B and Rodin to the filestore. VSRNet, ABZ 2008 (2008)
9. Butler, M.: Rodin deliverable D31: Public versions of plug-in tools. Technical report, University of Southampton, UK (2007)
10. Butterfield, A., Freitas, L., Woodcock, J.: Mechanising a formal model of flash memory. Science of Computer Programming 74(4):219–237 (2009)
11. Damchoom, K., Butler, M., Abrial, J.-R.: Modelling and proof of a tree-structured file system in Event-B and Rodin. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 25–44. Springer (2008)
12. Ferreira, M. A., Silva, S. S., Oliveira, J. N.: Verifying Intel Flash File System Core Specification. Technical report, University of Minho (2008)
13. Freitas, L., Woodcock, J., Fu, Z.: POSIX file store in Z/Eves: An experiment in the verified software repository. Science of Computer Programming 74(4):238–257 (2009)

14. Hoare, T., Misra, J.: Verified software: theories, tools, experiments; Vision of a Grand Challenge project (2005)
15. Hughes, J.: Specifying a visual file system in Z. Technical report, Department of Computing Science, University of Glasgow (1989)
16. Hynix Cemicondutor et al.: Open NAND Flash Interface Specification, Revision 2.0. Technical report, ONFI (Feb. 2008), http://www.onfi.org
17. Intel Flash File System Core Reference Guide, version 1. Technical report 304436001, Intel Coorporation (Oct. 2004)
18. Jackson, M.A.: *System Development.* Prentice–Hall, 1983.
19. Joshi, R. and Holzmann, G. J.: A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005)
20. Kang, E. and Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) First International Conference, ABZ 2008. LNCS, vol. 5238, pp. 294–308. Springer (2008)
21. Morgan, C. Sufrin, B.: Specification of the Unix filing system. In IEEE Transaction on Software Engineering, vol. 10, pp. 128–142 (1984)
22. Woodcock J, Davies, J. *Using Z: Specification, Refinement, and Proof.* Prentice–Hall, 1996.