UNIVERSITY OF SOUTHAMPTON

An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B

by

Kriangsak Damchoom

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

October 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Kriangsak Damchoom

Nowadays, many formal methods are used in the area of software development accompanied by a number of advanced theories and tools. However, more experiments are still required in order to provide significant evidence that will convince and encourage users to use, and gain more benefits from, those theories and tools. Event-B is a formalism used for specifying and reasoning about systems. Rodin is an open and extensible tool for Event-B specification, refinement and proof. The flash file system is a complex system. Such systems are a challenge to specify and verify at this moment in time. This system was chosen as a case study for our experiments, carried out using Event-B and the Rodin tool. The experiments were aimed at developing a rigorous model of flash-based file system; including implementation of the model, providing useful evidence and guidelines to developers and the software industry. We believe that these would convince users and make formal methods more accessible. An incremental refinement was chosen as a strategy in our development. The refinement was used for two different purposes: feature augmentation and structural refinement (covering event and machine decomposition). Several techniques and styles of modelling were investigated and compared; to produce some useful guidelines for modelling, refinement and proof. The model of the flash-based file system we have completed covers three main issues: fault-tolerance, concurrency and wear-levelling process. Our model can deal with concurrent read/write operations and other processes such as block relocation and block erasure. The model tolerates faults that may occur during reading/writing of files. We believe our development acts as an exemplar that other developers can learn from. We also provide systematic rules for translation of Event-B models into Java code. However, more work is required to make these rules more applicable and useful in the future.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, **Kriangsak Damchoom**, declare that the thesis entitled **An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B** and the work presented in the thesis are both my own, and have been generated by me as the result of my own research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- any part if this thesis has previously been submitted for a degree or any other qualification at the University of any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted form the work of others, the source is always given. With the exception of such quotations this thesis entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as

– K. Damchoom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, SBMF, volume 5902 of Lecture Notes in Computer Science, pages 134-152. Springer, 2009.

– K. Damchoom and M. Butler. An experiment in applying Event-B and Rodin to a flash-based filestore. In Rodin User and Developer Workshop, July 2009.

– K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and proof of a tree- structured file system in Event-B and Rodin. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, ICFEM, volume 5256 of Lecture Notes in Computer Science, pages 25-44. Springer, 2008.

Signed: .....

Date: .....

Acknowledgements

I would like to thank my supervisor, Prof Michael Butler, for his very valuable help and guidance. His supervision really helps me a lot in carrying out this research. Many problems would not be solved without his guidance. His ability really made me surprise many times when he could solve the problems that I had tried many times but failed. This thesis would not succeed without his kind supervision.

Many thanks to the Royal Thai Government and Department of Mathematics and Computer Science, Prince of Songkla University, for financial support and giving me a good chance to study abroad.

I would like to thank Dr Denis A. Nicole for his suggestion and valuable comment about the file system. Many thanks to Prof Jean-Raymond Abrial who gives me a good idea of modelling and proving of the tree properties.

I would like to thank my examiners, Prof Luc Moreau and Dr Andrew Ireland, for their valuable comment and suggestion that are really useful for making this thesis complete.

Many thanks also go to my mother Pim, wife Aom, sisters Joy and Ann, brother Dam, friends (Salang, Khampee, Supat, Somporn, Pakwan, Somphop, Tossaporn, Athitaya, Yaowaret, Pornrawee, Watsawee, Oat, Nam, Pundita, Onjira, Thanyalak, Sutat, Jarutas, Antony, Paiphan, Yuwapat, Waraporn and Num-Oratai) and colleagues (Devakar, Andy, Collin, Reza, Mar Ya, Nurlida, Lis, John, Renato, Shamim, Edward, Pasha and Ali) for their support and encouragement.

Kriangsak Damchoom

Dependable Systems and Software Engineering Group
School of Electronics and Computer Science
University of Southampton, UK

To my grand mother, Rom Damkaew

Chapter 1

Introduction

Over the past three decades, formal methods have been introduced and used in several areas of software development. Recently, the software industry has made use of VDM [84], Z [20], ASM [18], B [1] and Event-B [4]. However, research in this area is still going; attempting to make improvements and achieve more benefits from these methods. Work is also underway to bridge the gap between requirements, specifications and implementations.

1.1 Background and Motivation

In a grand challenge in verified software proposed by Hoare and Misra in [72], they state that *theories*, *tools* and *experiments* are three main areas of challenging research in formal verification. Nowadays, many advanced theories and useful tools are developed and used in the software community. In addition, the performance of modern machines (compared with the past decades) is great enough for the advanced computations needed for formal reasoning such as theorem proving and model checking. Experiments are also important and still needed for this discipline in order to push forward scientific progress in formal methods and make formal methods more accessible to software industries. Hoare and Misra say that experiments should be carried out by using existing tools and theories with selected areas of real-world systems especially for those systems concerned about safety or security. Experiments help us to understand the strengths and weaknesses of theories and tools. Experiments provide scientific evidence that can support the analysis of those theories and tools, and encourage other researchers to engage in more effective research in the future.

Event-B [4] is a formal method which is an extension of the B-method [1]. It is intended for specifying, and reasoning about, complex systems. These include concurrent and communicating systems [31]. Tool support for the Event-B method is provided by the

Rodin platform [6, 5]. This tool is based on Eclipse [57] and is designed to provide an extensible specification and verification environment.

A flash-based file system has been proposed as a challenging system by Joshi and Holzmann [85]. As presented in [85], there are some reasons why a file system is attractive: Firstly, a file system is complex enough even though it is only one part of an operating system. For example, how do we deal with failures that may occur while performing file or flash operations? How do we cope with fault-tolerance; when flash instructions fail or, with power loss? How do we ensure reliability in the presence of concurrent accesses? Moreover, correctness and security are very important for file systems, since important data, stored on modern machines, is now managed by the file system. Additionally, most current file systems have a well-established, well-defined, interface based on the POSIX standard[1] [85]. However, although fundamental data structures and algorithms used in the design of file systems are well-understood, file systems still have bugs. These pose a serious problem to users and enterprises. For example, in case of NASA, the software used for management of flash memory cards (in space missions) has a problem with unpredictable failures. This can give rise to sudden power-loss and reboots [76].

Three issues that should be addressed when doing research into file systems are functional requirements, underlying hardware and fault-tolerance [117]. Flash memory is an attractive option for implementing file systems because flash memory has no moving parts, consumes low power and is easily available. It is presently used in many kinds of storage devices. For example, flash memory has recently become a popular choice for nonvolatile storage used on spacecraft [85].

1.2 Research Direction and Goal

Direction: As mentioned earlier, an experiment is an important research approach in the field of formal methods. Thus, performing experiments with a formal method and tool was chosen as the direction of our research. Our work has been carried out by focusing on **experiments** using **Event-B** and the **Rodin** platform. A **flash-based file system** is selected as a case study for experiments. A goal of the experiments is to formally develop and implement a flash-based file system. In this work, several modelling techniques/styles are addressed in order to evaluate and compare. A set of functional requirements – such as operations affecting a tree structure, read and write operations – are selected as part of the system to be modelled, verified and implemented. Three important issues to be covered in our specification are fault-tolerance, concurrency and

[1] POSIX (Portable Operating System Interface) is a standard defining application programming interfaces (APIs) for file systems [66].

a wear-levelling process[2].

Research contributions: The research goal is to produce scientific evidence in five forms as follows:

(i) Verified models of a flash-based file system consisting of specification, refinement and proof. Our models cover three main issues: fault-tolerance, concurrency and wear-levelling. Concurrent file operations (e.g. read and write) were not covered in other related work. In addition, the structure of the file system and features we covered are also different from others. For example, we represent the filesystem structure as a tree structure using parent function, instead of using named-path. This work contributes to the grand challenge [85] already mentioned. (Details of comparison are given in Section 7.3.)

(ii) Several techniques – in modelling, refinement and proof – have been investigated and compared to choose appropriate ones for our development. An incremental approach has been selected as our main strategy. This approach was not used in other related work in modelling and verifying of file systems (as discussed in Chapter 7). We have followed a strong systematic refinement approach to organizing the refinement process that would be useful for other researchers and practitioners. Our work covers a full formal development (i.e. abstract specification, refinement, proof and implementation) that is rarely found in other work. Much existing formal modelling work usually ends with a specification/model without transforming it to an implementation. We believe that this would be of benefit to other people who are learning formal methods and/or carrying out research (especially in Event-B).

(iii) Systematic translation of Event-B models into Java code together with an implementation which is derived from the formal methods. The aim is to provide systematic translation rules and show that the models we specified are implementable by applying the rules we proposed. Our translation rules are defined for direct translation of Event-B models into Java code, which is different from others. For instance, Edmands et al [49] provide a tool to for specifying Java-like models, which are able to be translated into Event-B models for formal verification, and are able to be translated into Java code for implementation. (Related work is discussed in Section 8.4).

(iv) Assessments of methods and tools used for specification. Our experiment aims to assess Event-B and the Rodin tool by evaluating experimental results such as proof statistics, an implementation (whether the final outcome is satisfactory or not) and facilities of the tool (whether it is convenient for the users) compared

[2] A technique used for prolonging the life time of flash memory covering relocating and erasing blocks within a flash chip [62].

with other methods and tools. The assessments we propose in this thesis show the strengths and weakness of Event-B and the Rodin tool that would be useful for further improvement of the language and tool. For example, what extensions could be added to languages or tools in order to make them more useful and accessible in the future?

(v) Modelling, refinement and proof guidelines. These aim to provide new information/evidence that would be useful for formal developers, further research and the software industry – in terms of modelling techniques, styles or patterns used for formal specification, etc. For example, which modelling styles/techniques are suitable for the problem area? Which modelling styles can make proof simpler? How essential benefits can be achieved from existing theories and tools?

1.3 Methodologies and Results in Brief

Our experiments are carried out by using Event-B and the Rodin platform. A flash-based file system is chosen as our case study. An incremental refinement is employed as our strategy to develop a model of a flash-based file system. The refinement is used in two different approaches, horizontal or *feature augmentation* and vertical refinements or *structural refinement* [27]. Horizontal refinement is aimed at introducing new requirements or features which were not addressed in the initial model or may be postponed to other refinement steps. Thus, in each refinement step, additional state variables and related events might be added/extended to incorporate those features which are introduced. The system models will be enlarged gradually when new properties are added. On the other hand, the purpose of structural refinement 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 our development, we began with an initial model focusing on manipulation of a tree structure. After that, horizontal refinements were used to enlarge the model by introducing new features in refinement steps. We finally got several levels of a refinement chain representing a model of an abstract file system. After that, structural refinement was employed to relate the abstract file system with the flash specification. Event-decomposition was used in this step to decompose atomic events (file-read and -write) into sub-events in order to relate to the interfaces (page-read and -program) provided by the flash interface layer. Then, machine decomposition was employed to decompose the file system machine, that has already been replaced by the flash specification, into two sub-machines (representing the file system layer and the flash interface layer). At this point, we then have two sub-machines that can be further refined separately. In our work, further refinements focussing on the flash specification were carried out to cover

other flash features and the wear-levelling process.

Our development covers three main issues (i.e. fault-tolerance, concurrency and the wear-levelling technique). We have two main Event-B models in our development. One represents the file system layer and the other one represents the flash interface layer. The model of the flash interface layer provides interfaces (page-read and page-program) to the file system layer. The wear-levelling technique, a technique used for prolonging the lifetime of the flash devices, is specified in this model. The model of file system describes tolerance of faults that may occur at any point during reading/writing of a file. The model can also deal with concurrent file read and write operations. The flash interface model can also deal with concurrent page read/program and block-erase events, and faults.

In this work, we also have an evolution of the file system model described in Chapter 5. The evolution aims at revising the file system model to satisfy the requirements (i.e. partial read/write operations and unbounded version numbers) that have been changed. From this development, we outline the effect of this evolution and reusability of modelling and proofs.

Theorem proving is a methodology used for reasoning about our models. For all developments in Chapter 5 and 6, 1069 POs (Proof Obligations) were automatically generated by the Rodin tool. 671 POs were generated for the file system model and 398 POs were generated for the flash interface model. Most of them, 94% (of the file system model) and 100% (of the flash interface model), were discharged automatically (i.e. in total 1028 of 1069 POs (96%) were proved automatically). The rest, 41 POs, were proved interactively using the Rodin tool.

Based on experiences of modelling and proof, we provided some useful guidelines that developers may learn from. The guidelines are classified into three categories: modelling, refinement and proof. We also investigated and proposed systematic translation rules to translate Event-B models into Java code. The set of translation rules is divided into two parts: class construction and event transformation. However, future work is required to automate the application of these rules. Finally, we also implemented a prototype of a flash file system following the specification and the set of translation rules we proposed. This implementation covers two parts: a file system layer and a flash interface layer. We simulated part of the flash interface layer instead of using the real flash because we want to be able to simulate faults and test whether our model can deal with that.

1.4 Chapter Outline

In Chapter 2, we outline some existing formal methods together with reasons for their importance for software engineering. Chapter 3 details Event-B and the Rodin platform.

In Chapter 4, Chapter 5 and Chapter 6, work undertaken with case studies – a tree-structured file system and the flash memory – are used to show how to specify and refine system models using Event-B and Rodin. Related work on flash file systems are discussed and compared in Chapter 7. Systematic translation rules for translating Event-B models into Java code are proposed in Chapter 8. An implementation of a flash file system is outlined in Chapter 9. Modelling, refinement and proof guidelines are discussed in Chapter 10. Finally, a conclusion, assessment of Event-B and Rodin, and future work are given in Chapter 11.

Chapter 2

Formal Methods

2.1 Introduction

The purpose of this chapter is to outline an overview of formal methods, techniques and tools used for specification and verification. This chapter begins with giving a definition of formal methods in Section 2.2. Secondly, reasons why formal methods are important for software engineering are outlined in Section 2.3. Thirdly, some existing formal methods and tools used for specification and verification are given in Section 2.4. In this section, Z [20], B [1] and VDM [84] are chosen as examples of formalisms to be outlined and compared. The reasons we chose them are (i) they are state-based approach, which is an underline approach of Event-B and (ii) they are methods recently used in the software industry. However, there are other formal methods recently used such as ASM [18] and Alloy [81] which are not explained here. Other formalisms such as temporal logic, process algebras and action systems are briefly described in Section 2.5. Finally, refinement technique will be given in Section 2.6.

Note: To make it easier for readers to follow, Event-B – which is the method used in our development – will be outlined separately in Chapter 3.

2.2 Formal Methods

A definition of formal methods can be defined as mathematically-based techniques used for specifying, verifying and reasoning about software and hardware systems [1, 32, 84]. Formal methods are intended to explain software systems to both users and developers with a precise documentation which is structured and presented at an appropriate level of abstraction [83]. In addition, formal methods are aimed at providing users mechanisms, such as automatic provers and model checkers, to verify models.

Because of employing mathematical notation to specify systems, models specified by using formal methods are well-formed statements in mathematical logic that can be verified by mathematical processes. Moreover, the value of formal methods is that they provide a means for users (designers or developers) to construct a precise model of the system which is later to be implemented. The model is not the system itself. It is an abstract representation of the real system, allowing reasoning about the system without having it at ones disposal yet [1]. This means that the model cannot be tested or executed to verify that the model works properly and has properties that satisfy our needs. Similarly, we cannot use any room inside a model of a building. Therefore, reasoning about it is a powerful way to analyse a model [6]. Formal specification languages support specification of what a system should do. In contrast, programming languages are designed for specification of how results should be achieved. Although functional programming languages, such as ML (which stands for "Meta Language") [106, 69], Haskell [77, 79], and Scheme [55], are more like specification languages since these describe what result is expected, they are designed to be executable [20].

2.3 Why Formal Methods are Important for Software Engineering

The following reasons are summarised from Bowen [20] and Holloway [73] in order to describe why formal methods are important for the software development process.

As mentioned earlier, a formal specification is a well-formed mathematical statement. Because of its precision, even if such a specification is invalid – for example, the specification is not what the customer expected – compared with an informal specification, it is easier to tell where and why it is incorrect and fix it [20]. For example, when we find some things that go wrong during a development process we can go back to see the specification components such as invariants, preconditions or proof obligations in order to check whether they meet the requirements or not and fix them. In contrast, an informal specification is often ambiguous, it is difficult to find errors and eventually fix them. Additionally, employing mathematical notation increases the understanding of the behaviour of a system, particularly early in a design phase. It can aid designers to organise their thoughts, and make a model clearer, simpler and easier to understand [20].

Moreover, formal reasoning about a system is possible by stating and proving theorems about it. These provide a mechanism to check whether the system behaves in the way as we expected or not. Formal methods also help developers in reasoning about the operation of the system before its implementation [20]. For example, preconditions of each operation can be checked to see whether they satisfy the requirements either by manual inspection or using tools for model checking and animation.

The presence of design flaws is a major reason why software can go wrong or does

something which is not what we expected. Therefore, to ensure that the software system does what it is intended to do, design flaws must be handled in some way. Even though there are some different approaches used to handle the design flaws such as testing, design diversity, and fault avoidance, a suitable way that can reduce the design flaws is avoidance by using formal methods [73]. For small systems or systems with low reliability requirements, testing may be possible to show that the system meets its requirements. However, for high integrity software systems, such testing would require much more time than is feasible. Importantly, a test-based approach cannot cover the cases outside test cases applied to the system domain. Namely, an error may occur when the system tries to execute some cases outside that might be reached in the execution. Thus, for those systems, testing-based approaches are inadequate.

As stated in [20], a precise specification specified by using formal methods is easy to be followed until an implementation phase. The possible errors in a design can be reduced. Consequently, when errors could be found and fixed at the design phase, the number of iterations through a development cycle could be reduced.

Another point is that development cost is critical. If flaws could be found at the design stage, it would be cheaper to fix them than if they are found later in testing process [20].

2.4 Existing Modelling Languages

In this section, some existing specification languages including Z, VDM and B will be outlined in order. At the end of this section, a comparison among them will be given.

2.4.1 Z Notation

The Z notation [20] is a formal modelling language used for describing and reasoning about computer-based systems. It is aimed at providing precise specifications of systems and formulation of proofs about intended system behaviour. It was originally introduced by Jean-Raymond Abrial in the late 1970s and later developed by members of Programming Research Group at Oxford University [20].

Bowen [20] states that all expressions specified in Z notation are based on standard mathematical notations used in set theory, lambda calculus, and first-order predicate logic. Z contains a standardized list of mathematical functions and predicates which are commonly used in specification.

The problem with using mathematics alone is that large specifications usually become unmanageable and unreadable. Hence, a schema notation is included in Z to aid the structuring of specifications. This provides a framework for a textual combination of

sections of mathematics (known as schemas) using schema operators which are similar to the mathematical operations[20].

The concept of an abstract specification in Z is to specify what a system does rather than how the system does it. It is designed to be expressive and easy to reason about by humans rather than executable by computers.

2.4.2 Z Structure and Example

Schemas, represented using box notation, are introduced to aid the structuring of Z specifications. The schemas are used to describe two main parts of a specification: *state space* and *operations* [20].

The state-space schema shown below is divided into two parts, the first part is used to define the state variables *(x1, x2, ..., xn)* and the second part is an area for specifying invariants of those variables.

StateSpace

$$x1 : S1$$
$$x2 : S2$$
$$\ldots$$
$$xn : Sn$$

$$Inv(x1, ..., xn)$$

Below is an example of a Z schema of the visual file system model introduced by Hughes [78]. The schema shown below is a state space of *FileSysState* which consists of a set of objects named *objects* and function *parent*. The *parent* is specified as a partial function mapped from OBJ to OBJ, where OBJ is a set-type. Two constants, *desktop* and *trash*, are specified in this schema.

FileSysState

$$objects : \mathbb{P}\, OBJ$$
$$loc : OBJ \pfun OBJ$$

$$desktop \in objects$$
$$trash \in objects$$
$$\mathrm{dom}\, parent = objects - desktop$$
$$\mathrm{ran}\, parent \subseteq objects$$
$$parent(trash) = desktop$$
$$(parent^{-1})^*(\{desktop\}) = objects$$

Those invariants given above show that (1) *desktop* and *trash* are elements of *objects*; (2) all objects except *desktop* have a parent; (3) the set of all parents is a subset of or equal to set *objects*; (4) the parent of *trash* is *desktop*; and (5) all objects can be reached from *desktop*. The *asterisk (*)* represents a transitive closure. That is, $(parent^{-1})^*(\{desktop\})$ returns all objects that can be reached from *desktop*.

All operations in Z are considered to be atomic and can be structured in the following general way[20].

Operation

$x1 : S1;\ ...;\ xn : Sn$
$x1' : S1;\ ...;\ xn' : Sn$
$i1? : T1;\ ...;\ im? : Tm$
$o1! : U1;\ ...;\ op! : Up$

$Pre(i1?, ..., im?, x1, ..., xn)$
$Inv(x1, ..., xn)$
$Inv(x1', ..., xn')$
$Op(i1?, ..., im?, x1, ..., xn, x1', ..., xn', o1!, ..., op!)$

In the operation schema, $i1?, ..., im?$ are inputs, represented by including the $?$ symbol in the variable name while the outputs indicated by $!$ are $o1!, ..., op!$. The precondition is: $Pre(i1?, ..., im?, x1, ..., xn)$. And the state change $(x1, ..., xn)$ to $(x1', ..., xn')$ is specified by: $Op(i1?, ..., im?, x1, ..., xn, x1', ..., xn', o1!, ..., op!)$.

For example, the operation schema illustrated below represents the *Move* operation specified by Hughes [78]. An *obj* is an object to be moved to a new parent named *to*. Both *obj* and *to* are specified as elements of *OBJ*. They are identified as input variables.

Move

$obj? : OBJ$
$to? : OBJ$

$obj? \notin \{desktop, trash\}$
$to? \notin (parent^{-1})^*(\{obj\})$
$parent' = parent \oplus \{obj? \mapsto to?\}$

The invariants state that: (1) object *obj* must not be *desktop* or *trash*; (2) the target location or new parent, *to*, must not be an element of objects which are descendants of the *obj*, where $(parent^{-1})^*(\{obj\})$ returns all descendants of *obj*. Finally, the last invariant shows that the parent of *obj* will be changed to be *to*, where the oplus notation, $\oplus$, is a relation overriding.

2.4.3 VDM

VDM (Vienna Development Method) is one of the earlier formal methods introduced by a research group of IBM laboratory in Vienna. The aim of this method is to be used for writing system specifications together with discharging of proof obligations. These proof obligations are proved to ensure that the specifications maintain invariants [47]. VDM provides a framework for reasoning about the system specifications such as data types, operations, etc. All specifications and proof obligations are written in terms of predicates. HOL [64] is a theorem prover used for verification. VDM uses a special three-valued logic to deal with undefinedness – a predicate that cannot be identified as either true or false [105] – instead of classical two valued logic [84]. Even though VDM is not as popular as Z, it provides features of composition and decomposition [94].

2.4.4 VDM Structure and Example

VDM uses a module notation which is a combination of data definitions, state variables and a collection of operations to specify a system. The structure of the module is shown in Figure 2.1 [84].

```
module MODULE_NAME
...
definitions types
...
state
...
end ;
functions
...
operations
...
end MODULE_NAME
```

FIGURE 2.1: A module structure of VDM

Figure 2.2 shows an example of a VDM specification of a simple file system focusing on read and write operations. The module shown in Figure 2.2 is named *FILE*. This module has two parameter types: *FID* and *CONTENT*; and two operations: *WRITE* and *READ*. The definitions section states that *FCONT* is a function-type mapped from *FID* to *CONTENT*. The state of this file system is represented as *files*, which is typed as *FCONT* and is initialised to the empty set, where $files_0$ is an initial state.

WRITE is an operation that has an effect of writing a new file with a content to the file system. This operation uses an external state variable named *files* for writing, this

module *FILE*
parameters types $FID, CONTENT$
exports operations
$WRITE : FID \times CONTENT \rightarrow$,
$READ : FID \rightarrow CONTENT$
definitions
types
$FCONT = FID \rightarrow CONTENT$
state
State of *files* : $FCONT$
init $(mk\_State(files_0)) \mathrel{\widehat{=}} files_0 = \{\}$
end;
operations

$WRITE(i : FID, cnt : CONTENT)$
ext wr *files* : $FCONT$
pre $i \notin \mathrm{dom}\, files$
post $files = files^{\sim} \cup \{i \mapsto cnt\}$

$READ(i : FID)\ ocnt : CONTENT$
ext rd *files* : $FCONT$
pre $i \in \mathrm{dom}\, files$
post $ocnt = files(i)$

end *FILE*

FIGURE 2.2: An Example of VDM Specification

means that the *files* will be updated by this operation. There are two parameters used in this operation: i (an *FID*) and *cnt* (a *CONTENT*). The pre-condition states that i must not be an element of the domain of *files*, namely, this identifier must not have already been stored in the file system. The post-condition indicates that the state variable *files* will be equal to the previous state unions the new entry $(i \mapsto cnt)$ which is being added.

Operation *READ* is aimed at reading the content of an existing file. In this operation, i is an input parameter and *ocnt* is an output parameter. Here *files* is defined as an external state used for reading (since it is specified as *rd*). That is, changes are not allowed to be made to this state variable. The pre-condition, $i \in \mathrm{dom}\, files$, states that the given file ID, i, must exist. The post-condition shows that *ocnt* is equal to *files(i)*, which is a file content corresponding to the identifier i.

2.4.5 B-Method

The B-method [1, 35], originally developed by Jean-Raymond Abrial in the mid 1980s, is a state-based method used for specifying, reasoning about and coding software systems.

It is based on set theories which are used for data modelling, while generalized substitutions are used for describing state modifications through machine operations. Machine invariants are specified by using predicate logic. Refinement is used to relate models at varying levels of abstraction, and there are a number of structuring mechanisms (machine, refinement, implementation) used for organising a development.

The B-method is based on a notion of abstract machine and a notion of refinement. Variables of an abstract machine are typed by using set theoretic constructs such as sets, relations and functions [30]. The concept of refinement is the key notion for developing B models of computer-based systems in an incremental way. B models are accompanied by mathematical proofs that justify them. Proofs of B models convince the user (designer or specifier) that the models preserve all invariants and satisfy all refinement obligations [36]. The B-method has been selected as a tool by industries in area of critical systems concerning about risk. A notable example of the application of B is its industrial use in the railway control system in Paris (The Paris Metro) which has been working since 1998 [30]. Another example is the driverless at Paris Roissy Airport that has been operational in 2007 [3].

2.4.5.1 B Structure

The Figure 2.3 shows the structure of an abstract machine in classical B which consists of clauses: MACHINE, SETS, CONSTANTS, PROPERTIES, VARIABLES, INVARIANT, INITIALISATION and OPERATIONS [110].

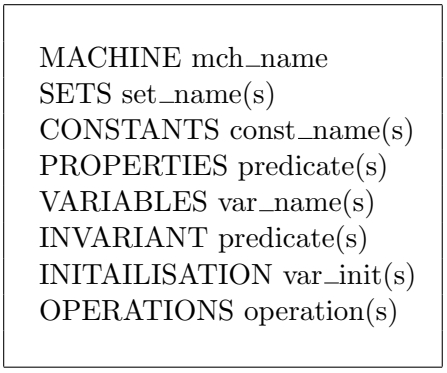


FIGURE 2.3: An Abstract Machine in B

The **MACHINE** clause defines a name of an abstract machine. In the example shown in Figure 2.4, the machine is named *Counter*. The **SETS** clause specifies all sets (types) used in the machine. The **CONSTANTS** clause identifies the constants which are used in the machine. The **PROPERTIES** clause describes the properties of those constants and sets. Considering the example given in Figure 2.4, there is one constant named *max*. Constant *max* is defined as a natural number ($\mathbb{N}$). The **VARIABLES** clause introduces all machine variables used in the machine. The **INVARIANT** clause

details all information related to the properties of the variables that must always be true such as types of those variables, relationships between the variables and their constraints or other restrictions on their values. All variables must have their types given in the Invariant clause. This means that there is at least one invariant clause for each variable which is defined in the Variables clause. The values of those variables can be changed when the machine is executed, however, such changes must not violate the invariants. The **INITIALISATION** clause is used to initialise the values of all variables of the machine. These values can later be modified by operations. The **OPERATIONS** clause specifies all operations required in the machine. The operations clause is used to describe the dynamic/behavioural properties of the systems.

2.4.5.2 An example of B-Specification

Figure 2.4 is an example of a specification in standard B where *Counter* is a machine name; *ctr* (counter), a machine variable, is initialised to zero; and *max* is a constant used to identify the maximum value of *ctr*. There are three operations in this machine: *incr* (increase the value of *ctr* by 1 at a time when this operation is performed); *decr* (decrease the value of *ctr* by 1 at time); and *display* is an operation for displaying the value of *ctr*. *PRE* clause identifies a precondition of operations. All actions within a THEN–END block will be performed only when the precondition holds.

2.4.6 A Comparison

Z, VDM and B are state-based formalisms in which a system is modelled by explicitly giving the definition of states and operations. Operations have an effect of transforming the system from a state to another state. In this approach, there is no explicit representation of concurrency.

Focusing on the structure of specification, although those methods have their own structure, they still have some parts which are similar in purpose. For instance, they all have an operation part and state variables.

The second point, focusing on operations, is that input and output variable are clearly defined in Z and VDM. Z uses "?" and "!" for input and output, respectively, while VDM uses "rd" and "wr" to classify the variables which are used for reading and writing, respectively. Although they all use the operation part to transform a system state to another state, their styles of transforming the system state are different. Namely, VDM [84] uses precondition and postcondition as a mechanism for specifying a process that aims to transform a state of program/model from one state to another state. The program must be performed in a state satisfying precondition and terminated in a state satisfying the postcondition. Z uses prime variables as post-state variables after the

MACHINE Counter
CONSTANTS
max
PROPERTIES
$max \in \mathbb{N}$
VARIABLES
ctr
INVARIANT
$ctr \in \mathbb{N}$
$ctr \leq max$
INITIALISATION
$ctr := 0$
OPERATIONS
incr =
PRE
$ctr \leq max$
THEN
$ctr := ctr + 1$
END
decr =
PRE
$crt > 0$
THEN
$ctr := ctr - 1$
END
$rst \leftarrow$ **display** =
BEGIN
$rst := crt$
END

FIGURE 2.4: An Example of B-Specification

change. Event-B uses generalized substitutions for transforming the model from one valid state to another valid state [4]. Namely, it must be proved that the substitutions that have been made to the state variable do not violate the desired properties that have been specified as invariants.

2.5 Other Formalisms

2.5.1 Temporal Logic

Temporal logic [95, 96] is a formalism for specification and verification of reactive systems. It has been used to describe and reason about behaviour of the systems which are concerned about time. In a temporal logic, a truth value of statements/propositions can

vary in time. This means that the truth value can be changed when the time changes while a truth value of the propositions in classical logic always be the same. In addition, temporal propositions generally contain some references to time conditions, while the classical logic deals with timeless propositions [89].

Temporal logic of actions (TLA) is a logic introduced by Lamport [91, 92]. It combines temporal logic with a logic of actions. TLA is used to specify and reason about concurrent and reactive systems by providing a mathematical foundation for describing the behaviour of the systems.

2.5.2 Process Algebra

Process algebra [13] is an algebraic approach used for describing or specifying behaviour of systems, especially for concurrent systems. This approach provides mathematical mechanisms and techniques to specify systems in terms of how processes interact, communicate, and synchronise with each other. The behaviour is the overall events or actions that the system can perform, and the actions are regarded as discrete, namely, concurrence may occur instantaneously.

There are many process algebras such as CSP (Communicating Sequential Processes) introduced by Hoare [71], CCS (Calculus for Communicating Systems) introduced by Milner [98] and ACP (Algebra for Communicating Processes) proposed by Bergstra and Klop [17]. Moreover, π-Calculus, originally developed by Milner [99], is an evolution of CCS to model concurrent systems consisting of mobile processes whose configuration is changing [50]. Although there are some differences between those methods, they use algebraic expressions and laws provided to describe and reason about the behaviour of communicating processes [22].

2.5.3 Action Systems

The action system formalism [12, 11], introduced by Back et al., is a state-based formalism for distributed systems. It provides a method to design the distributed systems that concentrates on the overall behaviour of the systems. The behaviour is defined in terms of possible actions that the processes can engage in, rather than in terms of a sequential code that the processes execute.

In case of process communication, Back et al [12, 11] also states that action systems provide a mechanism for processes to communicate or interact with each other during the execution. For example, when each process executes a sequential piece of code, it may communicate with the other processes by sending and receiving messages through shared variables or communication channel provided by the systems.

Additionally, an action system may be decomposed into a set of parallel sub-systems for implementation in a distributed fashion by breaking up the actions into sub-systems and using shared variables as a communication channel for interaction between those parallel action systems [23].

2.6 Refinement

Refinement [47, 8] is a mechanism that allows developers to sharpen their models step by step by adding more features or design details. Refinement aims at converting an abstract model into a concrete model that is implementable. As stated in [47], the main principle of refinement is that if the initial specification is valid and the refinement steps preserve correctness, then the resulting implementation will be correct by construction.

The refinement calculus is a calculus of program transformation. It provides rules for transforming abstract program structures to more concrete program structures while maintaining desired properties [37]. As stated in [28], originally, the refinement calculus was developed for sequential programs and then was extended to deal with distributed and parallel program via the action system of Back [11]. As stated in [116], it is also redeveloped individually by Morris [102], Morgan [100], and Back et al [10].

The concept of rule-based style of refinement (e.g. the refinement style of Morgan [100]) is to apply the rules to transform program fragments/models (S) from one form to another form (S'), automatically. $S \sqsubseteq S'$ (S is refined by S') holds if only if S' satisfies all desired properties that S satisfies [28]. Automatic transformation is a way that guarantees refinement [24].

Posit-and-prove is an alternative approach for refining models/programs. The concept of this approach is to rewrite a concrete model from the abstraction and then prove that the concrete one is the correct refinement of the abstract one (using theorem provers or model checkers) [24]. This is in contrast to the rule-based approach that is aimed at applying the rules to transform the abstract models/programs into concrete ones.

The refinement style used in Event-B follows the posit-and-prove approach. VDM also uses the posit-and-prove style for its refinement mechanism [39]. As already mentioned, by following this approach, an abstract model will be refined by rewriting it as a concrete one (without applying any transformation rules). Then, proof is required to show that the concrete model is the correct refinement of the abstract one. Desired properties of the model are specified as invariants (predicates formulated from state variables) that must be true forever. Such changes that have been made to the state variables (by events/operations) must be proved that those properties are maintained.

Refinement might be used in two different purposes that can be identified as follows:

Horizontal Refinement or Superposition Refinement

The purpose of this approach is to introduce new requirements or properties which are not addressed at the initial level or may be postponed to the next level. Thus, in each refinement step, additional state variables and events might be added to satisfy those requirements. The system models will be augmented gradually when new features of properties are introduced. This kind of refinement may be called *feature augmentation* [27]. For example, in case of a file system, an abstraction may start with introducing only functional requirements affecting a tree structure such as create, delete, copy, and move objects (files or directories) in the tree structure. The next refinement may add other requirements related to an object's properties such as file contents. Thus, in this refinement, some variables and events associated with this property need to be added to the model, such as variable *file-content* and events *read* and *write*. Similarly, other refinement steps may introduce other properties or events to satisfy other new requirements that may later be covered, such as the owner of each object and access permissions. Examples of this approach can be seen in Chapter 4, where horizontal refinement is used to introduce file system features in Section 4.4 up to 4.7.

Vertical Refinement

The aim of vertical refinement is to refine an abstract model by adding design details in each level of refinement down to an implementation. These refinement explain how features are achieved. Introducing new functional requirements or new properties is not appropriate for this approach. This kind of refinement may involve data and event/operation refinement, such as replacing an abstract state variable by a concrete one, breaking up an atomic event into sub-events, etc. To understand more about this approach, some examples of this refinement are given in Section 4.8. These examples are refinements of a file system focusing on read and write operations. Namely, at the abstraction, we begin with introducing abstract events which are later refined by being broken up into sub-events through refinement steps.

Chapter 3

Event-B

3.1 Introduction

Event-B [4, 6] is an extension of the B-method for specifying and reasoning about systems. Butler [31] states that Event-B was inspired by action systems of Back et al [12], which was described in the previous chapter. Event-B is an event-based approach which is defined in terms of a few simple concepts describing a discrete event system and proof obligations that permit verification of properties of the event system.

This chapter begins with describing the structure of an Event-B model which consists of two main parts: contexts and machines, in Section 3.2. Refinement in Event-B is described in Section 3.3. Other modelling techniques/features used in Event-B, i.e. event-decomposition, machine decomposition, event-extension and projection function for modelling record-types, are detailed in Section 3.5, 3.6, 3.7 and 3.8, respectively. The Rodin tool, a tool developed for Event-B modelling and verifying is introduced in Section 3.9. Finally, a comparison between Event-B and other state-based formalisms is given in Section 3.10.

3.2 Event-B Structure

An Event-B model [68, 97, 6] is described in terms of contexts and machines (*machine* is called *model* in [97]). Contexts contain the static parts whereas machines contain the dynamic part of a model. Contexts can be extended by other contexts and referenced by machines. Each machine can be refined by other machines. This structure is illustrated in Figure 3.1.

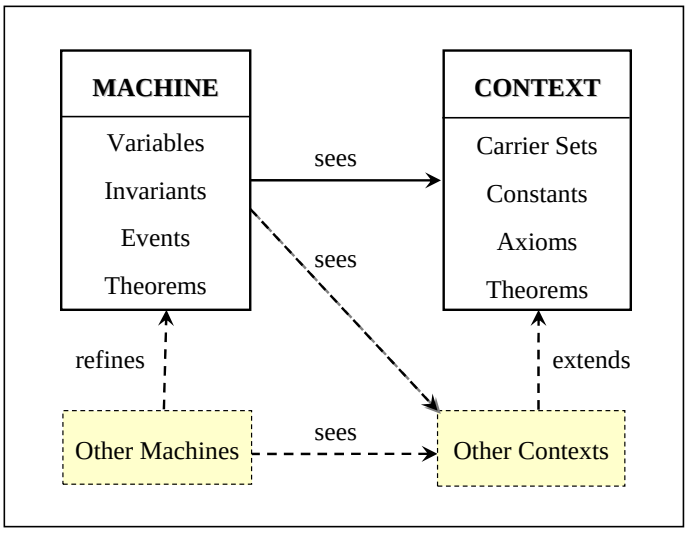


FIGURE 3.1: Relationships between machines and contexts

3.2.1 Contexts

Contexts [6, 8] contain the static parts of a model. Each context may consist of carrier sets and constants as well as axioms which are used to describe the properties of those sets and constants. Contexts may contain theorems for which it must be proved that they follow from the preceding axioms and theorems. Moreover, contexts can be extended by other contexts and seen by more than one machine. Additionally, a context may be indirectly seen by machines. Namely, a context C can be seen by a machine M indirectly if the machine M explicitly sees a context which is an extension of the context C.

3.2.2 Machines

Machines [6, 97] contain the dynamic part of an Event-B model. This part is used to specify behavioural properties of the model. A machine is made of a state, which is defined by means of variables, invariants, events, theorems and variants shown in Figure 3.1.

Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etc. They are constrained by invariants *I(v)* where *v* are the variables of the machine. Invariants are supposed to hold whenever variable values change. But this must be proved through the discharge of proof obligations [6].

A machine contains a number of atomic events showing the way that the model may evolve. Each event is normally composed of four elements: an event name, parameter(s), guard(s) and action(s), illustrated in Figure 3.2. The guard is the necessary condition for the event. The action determines the way in which the state variables are going to evolve when the event is performed [6].

Event
Name
Parameter
Guard
Action

FIGURE 3.2: Structure of an Event

All events are guarded and atomic and might be performed only when its guard holds. This means that when the guards of several events hold at the same time, then only one of them may be performed at that time. An enabled event is non-deterministically chosen to be performed. Generally, an event, named *evt*, is presented in one of three possible forms shown in Figure 3.3. *act* represents actions of an event that may involve generalized substitution of machine variables (v) and/or parameters (t) which are local to the event, while *grd* represents guards of an event. (Here t is called local variables in [6].)

evt $\widehat{=}$ **begin** *act(v)* **end**
evt $\widehat{=}$ **when** *grd(v)* **then** *act(v)* **end**
evt $\widehat{=}$ **any** t **where** *grd(t,v)* **then** *act(t,v)* **end**

FIGURE 3.3: Three syntactic forms of an event

There are three types of action illustrated in Figure 3.4: *skip* (do nothing), deterministic assignment and non-deterministic assignment. Where x is a variable, E is an expression and P is a predicate. The value of x in each case depends on its corresponding expression/predicate. For example, $x :\in E(t, v)$, x will be assigned as an element of $E(t, v)$. In the case of $x :| P(t, v, x')$, x will be assigned as a value satisfying the predicate P.

Type	Generalized Substitution
Empty	*skip*
Deterministic	$x := E(t, v)$
Non-deterministic	$x :\in E(t, v)$
	$x :\| P(t, v, x')$

FIGURE 3.4: Three types of generalized substitution

3.3 Refining a Machine in Event-B

Abrial et al [6] states that there are two possible ways of refining a machine, one is refining its state and another one is refining its events. Typically, both are used together.

In case of refining machine state – or data refinement [47] – gluing invariants play an important role to relate states of a concrete machine to abstract states. Gluing invariants

are invariants of a refined machine that refer to variables of the abstract machine [6, 5]. The gluing invariant is expressed in terms of a predicate $P(v, w)$ connecting the state variables of the abstract machine (v) and the corresponding state variables of the concrete machine (w) [5].

When refining events, each event of the abstract machine may be refined by one or more corresponding events in the refinement [6]. There are many cases where an event is considered to be refined. For example, when an abstract variable referred to in the event has been replaced by a concrete one (in a refinement step), some related guards and/or actions of that event may need to be changed. Namely, the abstract variable which is referred to in that event must be replaced by the concrete one. In the case of feature augmentation, for example, the abstract event may be extended by adding new features which are introduced in that refinement.

Adding new events in a refinement

During refinement, it is possible to refine an abstract machine by adding new events to its corresponding machine. The new events must be proved to refine a dummy event that does nothing (skip) in the abstraction. In this case, some proof obligations may fail to be proved if there are some actions of any new event trying to update a variable of the abstract machine. However, if necessary, a new variable (which is used as a mirror of the abstract variable at certain points) can be added to the concrete machine together with some gluing invariants relating the abstract variable with the new one. In addition, it may be proved that those events cannot collectively take control infinitely [6]. For this, as stated in [6], a unique variant expression has to be introduced. This variant will be decreased by those new events.

3.4 Event-B proof obligations

Several kinds of proof obligations are generated by the proof obligation generator (POG), such as WD (Well-definedness), INV (Invariant Preservation), GRD (Guard Strengthening), SIM (Action Simulation), etc.

WD proof obligations are generated to ensure that axioms, invariants, event guards/actions are well defined. The Rodin tool supports well-definedness to aid the activities of modelling and proving [5]. For example, as stated in [5], it can be guaranteed that partial functions are never applied to arguments outside their domain. INV proof obligations are generated to guarantee that the invariants are always preserved whenever the machine state changes. The generated GRD proof obligation ensures that the guard of a concrete event is a correct refinement of the corresponding guard of the abstract event. Finally, the generated SIM proof obligations aim to ensure that the abstract action are refined correctly by the action of the corresponding concrete event as specified by any

gluing invariants.

3.5 A technique for breaking up an atomic event

A technique for breaking up an atomic event into sub-events has been proposed by Butler and Yadav in [31]. This technique is based on the idea that an abstract (atomic) event may be realised by a number of activities or actions inside that can be split into sub-events through refinement steps. To understand more about event decomposition, event refinement diagrams proposed in [26] and a simple example will be used to explain how an atomic event can be decomposed into sub-events. Figure 3.5 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 been 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.

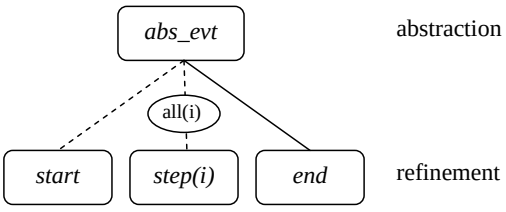


FIGURE 3.5: An example of event refinement diagram

In Event-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 state 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 [26], the diagrams were inspired by Jackson Structured Design (JSD) diagrams [82]. We see some useful of this systematic diagram. It would be great in the future if investigation for a more formal incorporation of event refinement diagrams into the refinement proof obligations could be carried out by developers who are interested.

To understand more about this technique, we give a simple example of breaking up an event which is discussed below.

The following example is aimed at breaking up one atomic event named *incr* (increase the value of x by the value of y) into sub-events named *start*, *step* and *end*; where x and y are integer variables. This example is divided into two levels: one abstraction and one refinement step.

Abstraction:

At this level, see Figure 3.6, an abstract atomic event named *incr* was introduced together with its failure case. Considering Event *incr*, the value of x is increased by the value of y. In the case of failure, the value of x is equal to a special value '*ko*'.

INVARIANTS	**INITIALISATION**
$y \in \mathbb{Z}$	$y :\in \mathbb{Z}$
$x \in \mathbb{Z}$	$x :\in \mathbb{Z}$
Event *incr* $\widehat{=}$	**Event** *incr_fail* $\widehat{=}$
Begin	**Begin**
$x := x + y$	$x := ko$
End	**End**

FIGURE 3.6: An abstract level

Refinement:

In this refinement, additional variables x', n and *flag* are added to the abstract machine. The variable n, a number of steps proceeded, is initialised to 0. Variable *flag* is a boolean variable used for checking whether or not the increasing step is completed. This flag is initialised to be FALSE. Machine invariants and its initialisation of this level are given in Figure 3.7.

INVARIANTS	**INITIALISATION**
$n \in \mathbb{Z}$	...
$x' \in \mathbb{Z}$	$n := 0$
$flag \in BOOL$	$x' := 0$
$flag = FALSE \Rightarrow x' = x + n$	$flag := FALSE$

FIGURE 3.7: Machine invariants and its initialisation of the concrete level

Event *step*, specified in Figure 3.8, is a sub-event which is added to specify that the value of x is increased by 1 at a time. This event shows that when the flag is equal to

FALSE (incrementing step has not completed yet) and $n < y$, x' and n are increased by 1. Event *end_ok* is a refinement of the *incr* event. This event states that when n is equal to y and the flag is FALSE, x will be assigned x' and the flag is set to be TRUE (indicating that the increasing step has completed). Gluing invariant of Figure 3.7 is used to discharge refinement proof obligations (SIM).

Event *start* $\widehat{=}$ **Begin** $n := 0$ $x' := x$ $flag := FALSE$ **End**	**Event** *step* $\widehat{=}$ **When** $n < y$ $flag = FALSE$ **Then** $x' := x' + 1$ $n := n + 1$ **End**
Event *end_ok* **refines** *incr* $\widehat{=}$ **When** $n = y$ $flag = FALSE$ **Then** $x := x'$ $flag := TRUE$ **End**	**Event** *end_fail* **refines** *incr_fail* $\widehat{=}$ **When** $flag = FALSE$ **Then** $x := ko$ $flag := TRUE$ **End**

FIGURE 3.8: Events of the concrete level

3.6 Machine Decomposition

Generally, a model is started with small number of features (small set of machine events and state variables) and then is enlarged gradually by adding more features or design details in refinement steps. Namely, machine variables and/or events might be added in each step. As stated in [8], the refinement process might become quite heavy if there are a large number of events and state variables. Moreover, it may be found that the refinement steps which are undertaken are not involving any more the totality of the system, that is only a few variables and events are concerned, while others are not important. Therefore, the idea of decomposition would be important for of formal modelling. The decomposition is a mechanism aimed at partitioning a large system model into smaller parts that can be addressed more easily than the whole. Namely, each part should to be refined independently of the others.

Two approaches have been proposed to the decomposition of Event-B models. The first is the shared variable decomposition which is proposed by Abrial et al [8]. The second is

the shared event decomposition proposed by Butler [26]. The difference between this two approaches is the method of the interaction between sub-models. The shared variable approach means sub-models interact with each other via shared variables, while the interaction of the shared event approach is the synchronisation over the shared events.

In our development, we follow the decomposition structures of Butler [26]. The decomposition structure given in [26] is a parallel-based decomposition. Namely, the machine variables and events are split into sub-machines. Each sub-machines must not have any common state variables. As mentioned above, each sub-machines interact with each other via the synchronisation over the shared parameterised events.

Figure 3.9 shows a scheme of an Event-B model named M. This machine consists of variables $v1$ and $v2$, and events $evt1$, $evt2$ and evt_s. Suppose we are decomposing this machine into two sub-machines (i.e. $M1$ and $M2$) as illustrated in Figure 3.10, where machine variables and events are split into $M1$ and $M2$. Namely, variable $v1$ and $evt1$ are placed in $M1$, while $v2$ and $evt2$ are placed in $M2$. Event evt_s is a shared event which is used for synchronisation. This shared event is also partitioned into two sub-events located in both sub-machines. This shared event depends on both $M1$ and $M2$, since it has the effect of updating both variables $v1$ of $M1$ and $v2$ of $M2$. This is in contrast to $evt1$ and $evt2$, where $evt1$ depends only on $v1$ and $evt2$ depends only on $v2$. Guards and actions (of the shared event) on $v1$ and on $v2$ are clearly separated. Namely, $v1$ is referenced by $grd3$ and $act3$, while $v2$ is referenced by $grd4$ and $act4$, separately. Parameters $p1$, $p2$ and $p3$ are local to the shared event. $M1$ and $M2$ can

MACHINE M
Variables
$v1, v2$
Invariants
...
Events
$evt1 \mathrel{\widehat{=}}$ **any** p **where** $grd1(p, v1)$ **then** $act1(p, v1)$ **end**
$evt2 \mathrel{\widehat{=}}$ **any** p **where** $grd2(p, v2)$ **then** $act2(p, v2)$ **end**
$evt\_s \mathrel{\widehat{=}}$ **any** $p1, p2, p3$ **where**
$grd3(p1, p3, v1) \wedge grd4(p2, p3, v2)$
then
$act3(p1, p3, v1) \parallel act4(p2, p3, v2)$
end
End

FIGURE 3.9: Machine M before decomposition

be refined separately provided shared events and shared parameters are maintained.

Figure 3.11 shows a result of the machine decomposition that has been made to Machine M. The top represents Machine $M1$ with Variable $v1$ and events $evt1$ and evt_s1. The

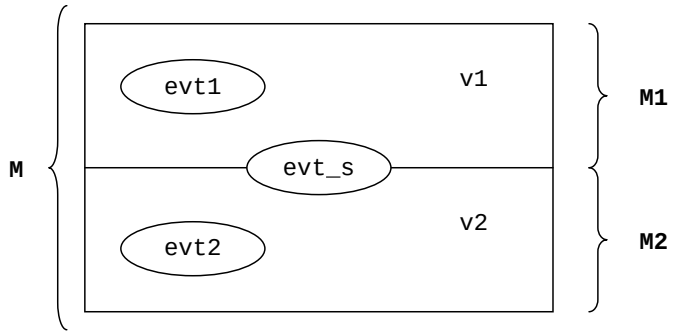


FIGURE 3.10: A diagram illustrating a decomposition made to Machine M

bottom represents Machine $M2$ with Variable $v2$ and events $evt2$ and evt_s2. Parameters $p1$ and $p2$ are local to evt_s1 and evt_s2 respectively, while $p3$ is a shared parameter across evt_s1 and evt_s2. This shared parameter is used for synchronisation of both sub-events.

MACHINE M1
Variables
$v1$
Invariants
...
Events
$evt1 \mathrel{\widehat{=}}$ **any** p **where** $grd1(p, v1)$ **then** $act1(p, v1)$ **end**
$evt\_s1 \mathrel{\widehat{=}}$ **any** $p1, p3$ **where** $grd3(p1, p3, v1)$ **then** $act3(p1, p3, v1)$ **end**
End

MACHINE M2
Variables
$v2$
Invariants
...
Events
$evt2 \mathrel{\widehat{=}}$ **any** p **where** $grd2(p, v2)$ **then** $act2(p, v2)$ **end**
$evt\_s2 \mathrel{\widehat{=}}$ **any** $p2, p3$ **where** $grd4(p2, p3, v2)$ **then** $act4(p2, p3, v2)$ **end**
End

FIGURE 3.11: Sub-machines representing a result of a decomposition of Machine M

3.7 Event Extension

Event extension is a feature that have been added to the Rodin tool release 0.9.x and later. The purpose of this feature is to make model easier to be refined (especially for

horizontal refinement). Namely, instead of repeating guards and actions of an abstract event in the concrete event, such events can be extended by introducing only part of specification that have been extended in that step. Instead of using **refines**, **extends** is used for modelling the event extension.

Figure 3.12 shows an example of event-extension. The top represents an abstract event named *crtfile_abs*, while the bottom represents the concrete event (*crtfile_ext*) which is an extension of the *crtfile_abs* event. In the concrete event, the *crtfile_abs* event is extended by adding a property, i.e. file content ($fcontent \in files \rightarrow CONTENT$, where *files* represents a set of existing files and $CONTENT = \mathbb{N} \nrightarrow DATA$ representing the content of files). Using the event-extension feature provided by the tool, that part of the specification which is inherited from the previous abstraction are omitted. Developers specify only part that have been introduced in that step. In this example, the extended part is $act3$, where the content of file is initialised to be empty. This feature also makes a

Event *crtfile_abs* **refines** *create* $\widehat{=}$
  Any *obj*, *in* **Where**
    $grd1 : obj \in OBJECT \setminus (files \cup directories)$
    $grd2 : in \in directories$
  Then
    $act1 : files := files \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

Event *crtfile_ext* **extends** *crtfile_abs* $\widehat{=}$
  Begin
    $act3 : fcontent(obj) := \varnothing$
  End

FIGURE 3.12: An example of event extension

model easier to be modified. Namely, some modification can be made to the abstraction and that change is automatically inherited by the refinement.

Figure 3.13 shows another style of representing an extended event. In order to make a difference between inherited part (*obj*, *in*, $grd1$, $grd2$, $act1$ and $act2$) and the extended part ($act3$) of the *ctrfile_ext* event, we represent the extended part in *italic* style while the inherited part is represented in normal style. This style makes documentation of extended events more understandable.

Event *crtfile_ext* **extends** *crtfile_abs* $\widehat{=}$
  Any *obj*, *in* **Where**
    grd1 : obj $\in$ OBJECT $\setminus$ (files $\cup$ directories)
    grd2 : in $\in$ directories
  Then
    act1 : files $:=$ files $\cup$ {obj}
    act2 : parent(obj) $:=$ in
    *act*3 : *fcontent*(*obj*) $:= \varnothing$
  End

FIGURE 3.13: A documentation style to represent an extended event

3.8 Projection Function for Modelling Records

Evans and Butler [53] have given an approach for specifying record types in Event-B using projection function. In order to model a record or a data type that may consist of two or more elements, using projection function is a way to specify this. Figure 3.14 shows an example of modelling of a record type following the style given in [53]. The record type is named RT, where the structure of this type is composed of two properties: $prop1OfRT$ and $prop2OfRT$. This record type is specified as a carrier set while its properties are specified as constants. Each property is defined using a projection function. The type of each property may be a user-defined type or a basic type such as $\mathbb{N}$, $BOOL$, etc. For example, the first property ($prop1OfRT$) of this record type is defined as a natural number while the second property ($prop2OfRT$) is defined as a user-defined type $T2$.

CONTEXT CTX
Sets
  RT, $T2$
Constants
  $prop1OfRT$, $prop2OfRT$
Axioms
  $axm1 : prop1OfRT \in RT \rightarrow \mathbb{N}$
  $axm2 : prop2OfRT \in RT \rightarrow T2$

FIGURE 3.14: Part of context specifying a record type

Suppose we have a machine variable named *mvar* (representing an array of RT elements) specified as

$$mvar \in \mathbb{N} \nrightarrow RT$$

Figure 3.16 gives an Event-B event showing the use of the record type (RT) which is

specified above. This event has the effect of modifying the value of *mvar* at position i to be *newval* ($act1$). The values of the components within the *newval* are $p1$ and $p2$, which are typed $\mathbb{N}$ and $T2$ respectively.

Event $modify\_evt \mathrel{\widehat{=}}$
  Any $i, newval, p1, p2$ **Where**
    $grd1 : i \in dom(mvar)$
    $grd2 : newval \in RT$
    $grd3 : p1 \in \mathbb{N}$
    $grd4 : p2 \in T2$
    $grd5 : prop1OfRT(newval) = p1$
    $grd6 : prop2OfRT(newval) = p2$
  Then
    $act1 : mvar(i) := newval$
  End

FIGURE 3.15: An event (*modify_evt*) showing the use of the record type RT

The record type (RT) which is defined in Figure 3.14 may be extended by adding more properties in another refinement step. Figure 3.16 shows an example of a context where the RT type is extended. This context (CTX2) is an extension of the context named CTX, which is given in Figure 3.14. The extension is to add an additional property named $prop3OfRT$ (which is $T3$) to RT.

CONTEXT CTX2 **extends** CTX
Sets
  $T3, ...$
Constants
  $prop3OfRT, ...$
Axioms
  $axm3 : prop3OfRT \in RT \rightarrow T3$
  ...

FIGURE 3.16: Part of context specifying a record type

Figure 3.17 shows an extended part of the *modify_evt* event when the additional property ($p3$) of RT has been added.

3.9 Rodin, an Event-B Modelling Tool

The Rodin platform [2, 29, 40, 5] is an open and extensible tool for Event-B specification and verification. This platform contains a database of modelling elements used for

Event *modify_evt* **extends** *modify_evt* $\widehat{=}$
  Any
    $p3$
  Where
    $grd7 : p3 \in T3$
    $grd8 : prop3OfRT(newval) = p3$
  End

FIGURE 3.17: A extension of the *modify_evt* event

constructing system models such as variables, invariants and events. It provides useful tools for users to specify their models, accompanied by flexible tools for refinement and proof. Abrial et al [6] express that extending the state of art in formal methods tools and allowing other developers to employ their tools as plug-ins to assist the development methods are the purposes of the kernel of the Rodin tool. It allows users to customize and adapt the primary tool to serve their particular need.

Several plug-ins are available for the Rodin platform [25], for example, UML-B [113], ProB [16], the decomposition plug-in [112], B2Latex [46]. These plug-ins have been developed to satisfy some features required for users who may want to animate their models (using the ProB animator) during the design or who may want to represent their models by using UML-like diagrams of UML-B. The B2Latex plug-in is a LaTeX code generator that we have developed to help users in translating their Event-B models into LaTeX documents. The shared-event composition [111] is another plug-in that was used in our development for machine decomposition. This composition plug-in is based on shared event decomposition of Butler [26]. The decomposition tool [112] is a recent one that has been developed to support both shared variable and shared event decomposition.

Theorem proving is the main technique used for reasoning about Event-B models. The Rodin tool supports automatic generation of proof obligations in order to free the users from difficult work of writing them explicitly [5]. Proving of the models will be attempted automatically whenever the model is saved. If some proof obligations have not been discharged automatically by the provers, the Rodin tool also provides a proof manager for users to carry out interactive proof. Other formal languages have theorem proving support: the Z/EVES system [109] has been used for Z; KIV theorem prover [14] has been used for ASM; and HOL [65] has been used for VDM. Z/EVES has a graphical interface and supports automatic type checking. However, users still need to construct proof scripts by hand. Similarly for KIV and HOL theorem provers, modellers are also need to construct all proof obligations by themselves.

The theorem prover provided by the Rodin toolset was chosen for the verification parts of our experiments. Although the Rodin toolset also supports animation and model check-

ing via the ProB plug-in [16], there were some reasons why we chose the theorem prover approach. First is the limitation of model checking caused by state space explosion. Model checking can guarantee correctness within a limited state space. It cannot ensure the correctness outside the given state space, e.g. the complex system with complex data structures that might involve a large state space. The theorem prover approach can reason about infinite state spaces and state spaces that involve complex data structures and recursion [103]. Theorem prover can reason about the model without visiting the state space by verifying logical properties of models. Our model is a complex one that results in a large state space for model checking. We had tried many times to use ProB plug-in but failed. At the earlier stage of modelling, where a small set of features had been introduced, the ProB model checker and animator worked well. But, when we refined the model by adding more design details, which made the data structures more complex, we were unable to use ProB for animating and checking our models. Thus, we decided to stop using it for verifying our models. Another reason, which is a main point, theorem proving approach helped us a lot in discovering of invariants. Failing proof obligations guided us to identify which invariants should be introduced. (It can be seen in our development outlined in Chapter 4, 5 and 6, where we discuss about this.)

3.10 A Comparison

As an extension of B, most of the notation used in an Event-B model such as sets, relations and functions are similar to B. Thus, developers who have used B for specification would find it easy to adapt to Event-B. However, there are some differences between B and Event-B. Firstly, the structures used to describe the model are different. The static part (context) and dynamic part (machine) are totally separated in Event-B. Secondly, Event-B is more suited to model complex systems such as distributed and concurrent systems. Because it is an event based approach which consists of a collection of guarded atomic events, a machine is viewed as a reactive system that continually executes enabled events in an interleaved fashion [30]. This makes parallel activities and concurrent processes easier to model as an interleaving of event executions, while shared variables/events are used for interaction between the activities/processes. Classical B is based on a passive model. Namely, operations are called by other operations. (In Event-B, an event is not necessary called by others.)

As stated in [26], Event-B refinement is more general than classical B and other related languages such as Z and VDM. The ability to introduce new events in a refinement step is an important feature of Event-B. Event-B refinement supports the decomposition of an atomic event and also the decomposition of a machine.

Composition/Decomposition of Event-B and classical B are different. Namely, Event-B uses A- and B-style (that have already discussed in Section 3.6) as a mechanism for

machine decomposition while classical B uses machine inclusion/import (which is based on program structuring).

Compared with other state-based approaches mentioned in Chapter 2, Event-B also uses generalized substitutions as a mechanism to transform a system state to another state like B. Event-B supports both concurrent and communication systems. Proof is a methodology used for verification of Event-B models similar to classical B.

Chapter 4

Modelling and Proof of a File System

4.1 Introduction

As previously mentioned in Chapter 1, a flash file system has been proposed as a challenge for verification technology and we have chosen it as a case study for our experiments. Figure 4.1 is a representative of a flash file system [67]. In the figure, the architecture was divided into two main parts. The first part, the dotted box, represents user/application and file system layers. The file system layer provides the generic interface to the file system itself. The second part (the dashed box) represents the flash file system core which is composed of the flash interface and hardware layers, and other intermediate layers such as *Data Object* and *Basic Allocation* layers (more details can be seen in [67]). This chapter presents a specification of the file system layer within the dotted box, focusing on basic functionalities of a tree-structured file system, and read and write operations. Details and specification of the flash interface layer within the dashed part will later be explored in Chapter 6.

The aim of this chapter is to investigate and describe how existing theories and techniques of specification, refinement and proof can be applied in Event-B and Rodin to the specification of a file system. For example, how horizontal and vertical refinements can be applied, how selection of formulation affects the specification and proof, how breaking up an atomic event and machine decomposition can be applied to this case study.

Incremental refinement is our main strategy in carrying out the work. We first use feature augmentation to incrementally specify a model of an abstract file system by adding new features in each refinement step. After that, structural refinements (covering event and machine decomposition) are used for adding more design details to relate the abstract file system to the specification of the flash interface. Here we get eight levels of

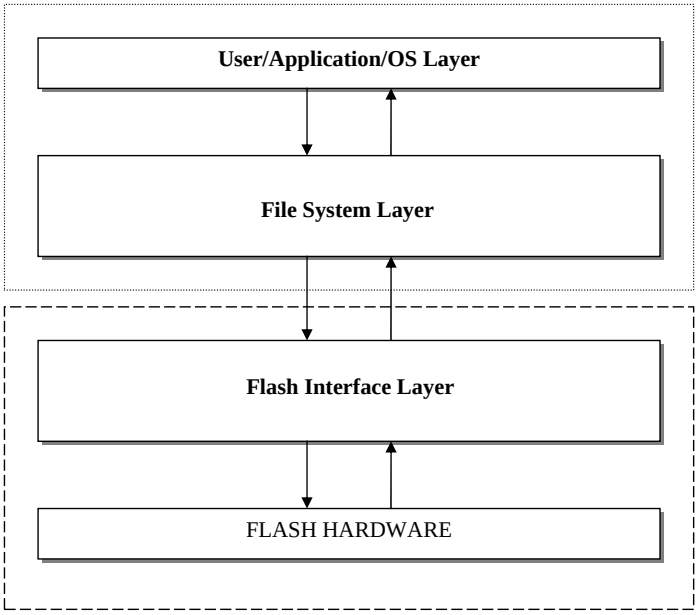


FIGURE 4.1: An architecture of a flash file system

specification modelling of a flash file system covering (1st) tree structure, (2nd) files and directories, (3rd) file content, (4th) permission control, (5th) other missing properties (name, creation date, etc.), (6th) decomposing file write (7th) decomposing file read and (8th) replacing by the flash specification. We split the features and choose to refine the model in this order for some reasons. First, we want to simplify the proof of the first level by postponing files and directories to be introduced in the following step. Second, file content should come after an introduction of files. Similarly for permissions, they should come after file content, since reading/writing of a file content depends on the permissions. Third, all file system features were covered before exploring structural refinements that involve adding more design details. Finally, we split the features into a number of refinement steps because we want to make the gap between each level as small as possible. We believe that the small gap leads us to get simpler gluing invariants and proofs. This testifies the proof statistics given in Table 4.1.

Our model covers concurrent file read/write operations. Several file read/write events can be performed simultaneously in an interleaved fashion. The model we developed also tolerates faults that may occur at any point during reading/writing of files. Details are given in Section 4.8.

This chapter begins with an informal description of a tree-structured file system and its constraints in Section 4.2. An abstract model of the file system layer and its horizontal refinements are given in Section 4.3 to Section 4.7. Vertical refinements which are explored to relate that file system layer with the flash interface layer are given in Section 4.8. The fault-tolerance issue is given in Section 4.11. Machine decomposition is outlined in Section 4.13. Full details of the specification are given in Appendix A. Proofs are given in Section 4.14. Finally, conclusion and assessment of what we have

achieved are discussed in Section 4.15.

Note: Much of the contents of this chapter appears in ICFEM 2008 [45], SBMF 2009 [44] and Rodin Workshop 2009 [43].

4.2 An Informal Description of a Tree-structured File System and Constraints

A tree-structured file system can be described in terms of a collection of objects representing files and directories and a set of operations that may be performed on these objects. The objects are structured as a tree. The tree has only one root directory that cannot be deleted, copied or moved. Each object except the root has only one parent which is a directory. Four operations affecting the tree structure are discussed below.

Create: Create an object in an existing directory. The object can be either a file or a directory.

Copy: Copy an existing object from one place to another place. The destination must exist and must not be a descendant of the object being copied or the object itself. If the object being copied is a directory, all objects belong to that directory must also be copied to the new location and the copy must have the same structure as the original.

Delete: Delete an existing object in the file system. In case of deleting a directory, all its descendants must also be removed.

Move: Move an existing object from one place to another place. The destination must exist and must not be a descendant of the object being moved or the object itself.

Note that the copy event we specify here is not traditionally found in file systems. Namely, the process of copy could be done by performing read and write operations provided. However, the copy operation is sometimes found at the higher level of user interfaces provided by operating systems, such as DOS and visual file system. It is also found in the specification of a visual file system in Z of Hughes [78]. The copy event is a complex event that directly affects the structure of the tree. Performing this event must not destroy the tree properties. The reason we have specified this operation in our model is to show that our abstract copy event preserves the tree properties.

4.3 An initial model

In our development, we begin with an abstract model of a tree-structured file system focusing on tree properties and operations affecting the tree structure. However, files and directories are not distinguished in this level. Instead, they are postponed to the

next refinement given in Section 4.4. Thus, in this level, both files and directories are treated in the same way as objects, which are nodes of the tree structure. Below is a list of requirements in this level.

Req1.1: The tree has a root node.

Req1.2: All objects except the root node must have a parent.

Req1.3: There are no loops in the tree.

Req1.4: Every node in the tree is reachable from the root node.

Machine variables, invariants which are formulated to satisfy those required properties mentioned above, and initialised values of those variables are given in Figure 4.2. Variables, invariants and initialisation are discussed below.

Variables
$objects$, $parent$
Invariants
$inv1.1 : objects \subseteq OBJECT$
$inv1.2 : root \in objects$
$inv1.3 : parent \in objects \setminus \{root\} \rightarrow objects$
$inv1.4 : \forall s \cdot (s \subseteq parent^{-1}[s] \Rightarrow s = \varnothing)$
Initialisation
$objects := \{root\}$
$parent := \varnothing$

FIGURE 4.2: Machine variables, invariants and initialisation of an abstract model

As can be seen from a context by this abstract machine, *OBJECT* is defined as a carrier set and *root* is an *OBJECT* constant (see Figure 4.4). Considering Figure 4.2, there are two state variables introduced in the machine: (i) *objects*, a set of existing objects in the file system ($inv1.1$); and (ii) *parent*, a total function mapped from all objects except *root* to their parent which is an object. In this abstraction, *objects* and *parent* are initialised to a set consisting of *root* and the empty set respectively. Invariant $inv1.3$ states that all objects except *root* must have a parent. This invariant satisfies $Req1.2$. Invariant $inv1.4$ is introduced to ensure that there are no loops in the tree structure (satisfying $Req1.3$). This invariant is formulated by using the no-loop property proposed by Abrial in [4]. The reason we choose this formulation instead of transitive closure which is generally used to specify tree properties – such as a specification of visual file system in [78] – is to make the model easier to prove.

Considering $inv1.4$, $parent^{-1}[s]$ gives the direct descendants of all elements of set s. For $s \subseteq objects$, $s \subseteq parent^{-1}[s]$ means that s contains a loop in the parent relationship.

Hence, this invariant states that the only such set that can exist is the empty set and thus the parent structure cannot have loops. If we were to use transitive closure, we would need to add the property $inv1.4b$ given in Figure 4.3 to the machine invariants.

$$inv1.4b : tcl(parent) \cap id(OBJECT) = \varnothing$$

FIGURE 4.3: No-loop property using transitive closure

Here tcl which is mentioned in Invariant $inv1.4b$ is a transitive closure. In a context shown in Figure 4.4, tcl is defined as a total function mapped from $OBJECT \leftrightarrow OBJECT$ to $OBJECT \leftrightarrow OBJECT$. Giving $r \in OBJECT \leftrightarrow OBJECT$, the transitive closure of r is equal to $r \cup r;\ tcl(r)$ ($thm1$ of Figure 4.4). The transitive closure we specify here follows from the definition given in [1].

The *parent* variable is updated by several of the events. If we were to use $inv1.4b$ instead of $inv1.4$, the *copy* event, for example, would give rise to a proof obligation with $inv1.4b$ as a hypothesis and the following goal:

$$tcl(parent \cup replica \cup \{nobj \mapsto to\}) \cap id(OBJECT) = \varnothing$$

The way to prove this proof obligation would not be easy since distribution of tcl through union and other set operations is not straightforward. We avoid such difficulty in proofs by using formulation $inv1.4$ instead.

Significantly, we can prove that the formulation in $inv1.4b$ follows from the formulation in $inv1.4$. This is given by Theorem $thm3$ shown in Figure 4.4. This theorem has been proved using the interactive prover of Rodin. The strategy we follow in proving this theorem is to use proof by contradiction.

In order to satisfy requirement $Req1.4$, instead of introducing another invariant, we present other machine theorems (given in Figure 4.5) which are derived from existing invariants and guarantee that the reachability property is satisfied. Considering Theorem $mth3$, since $(tcl(parent))^{-1}[\{root\}]$ returns all objects reachable from *root*, this theorem shows that all objects except *root* are reachable from *root*. Other machine theorems, $mth1$ and $mth2$, are used in the proof of $mth3$. Theorem $mth4$ is introduced to satisfy the no-loop property.

Abstract Events:

In this section, we outline four abstract events including *create*, *move*, *copy* and *delete*.

Create event: Create an object in an existing location (see Figure 4.6). In the figure, *obj* is an object being created and *in* is its parent. Here *obj* must be an OBJECT that

Sets
$OBJECT$
Constants
$root, tcl, objrel, objfn$
Axioms
$axm1 : root \in OBJECT$
$axm2 : objrel = OBJECT \leftrightarrow OBJECT$
$axm3 : objfn = OBJECT \setminus \{root\} \rightarrow OBJECT$
$axm4 : tcl \in objrel \rightarrow objrel$
$axm5 : \forall r \cdot (r \in objrel \ \Rightarrow \ r \subseteq tcl(r))$
$axm6 : \forall r \cdot (r \in objrel \ \Rightarrow \ r;\ tcl(r) \subseteq tcl(r))$
$axm7 : \forall r, t \cdot (r \in objrel \ \wedge \ r \subseteq t \ \wedge \ r;\ t \subseteq t \ \Rightarrow \ tcl(r) \subseteq t)$
Theorems
$thm1 : \forall r \cdot (r \in objrel \ \Rightarrow \ tcl(r) = r \cup (r;\ tcl(r)))$
$thm2 : tcl(\varnothing) = \varnothing$
$thm3 : \forall t \cdot (t \in objfn \ \wedge \ (\forall s \cdot s \subseteq (t^{-1})[s] \ \Rightarrow \ s = \varnothing)$
$\Rightarrow tcl(t) \cap id(OBJECT) = \varnothing)$

FIGURE 4.4: Definition of transitive closure (tcl) and no-loop theorem ($thm3$) in a context

Theorems
$mth1 : \forall T \cdot (root \in T \ \wedge \ parent^{-1}[T] \subseteq T \ \Rightarrow \ objects \subseteq T)$
$mth2 : objects \subseteq \{root\} \cup (tcl(parent))^{-1}[\{root\}]$
$mth3 : objects \setminus \{root\} \subseteq (tcl(parent))^{-1}[\{root\}]$
$mth4 : tcl(parent) \cap id(OBJECT) = \varnothing$

FIGURE 4.5: Machine theorems satisfying reachability and no-loop properties

is not already in the set *objects* (see $grd1$); and *in* must exist (see $grd2$). The object *obj* will be added to the set *objects* by $act1$; and *in* will be assigned to be the *obj*'s parent by $act2$.

Copy event: In order to understand more about the copy event, we will describe this event by using Figure 4.7. From the figure, the left-hand side is a tree before copying and the right-hand side is the result. Here r is a root node, a is an object being copied (d and e, its descendants, will be copied as well) from node r to node c. The arrows represent the function *parent* and the dashed lines represent a correspondence function which is a bijection from the set of all objects being copied to the set of new objects (a', d', and e') which is a copy of that set. The correspondence bijection is used to maintain the structure of directory a in the copy.

Considering the *copy* event given in Figure 4.8, *obj* (the object being copied) and *to* (the destination) behave like external parameters provided by users or application programs, while the rest are local parameters used for computation. However, there is no distinction

Event *create* $\widehat{=}$
  Any
    obj, in
  Where
    $grd1 : obj \in OBJECT \setminus objects$
    $grd2 : in \in objects$
  Then
    $act1 : objects := objects \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

FIGURE 4.6: A specification of create event

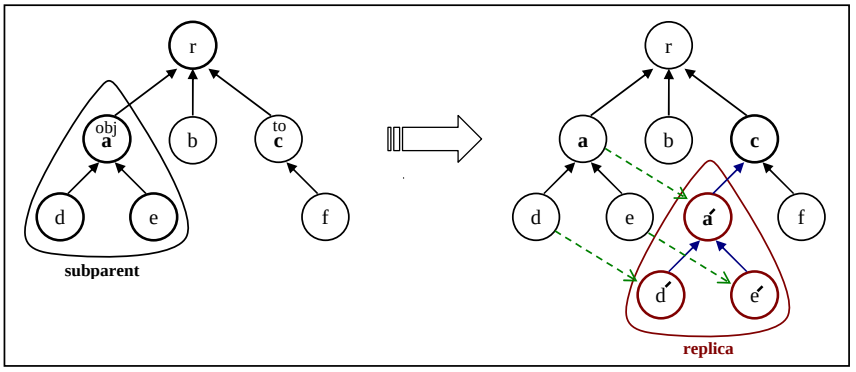


FIGURE 4.7: A diagram of copying a subtree (*subparent*) rooted at a from r to c

between external parameters and local parameters in Event-B. In this event, *des* is the set of all descendants of the object *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$; *objs* is the set of all objects being copied; *nobjs* is the set of new objects corresponding to the set *objs*; *corres* is the correspondence bijection. With reference to Figure 4.7, *subparent* represents the subtree rooted at a which is being copied. In this event, *subparent* is equal to $des \lhd parent$ which is a restriction of the parent function to *des* (e.g., $d \mapsto a$ and $e \mapsto a$ in Figure 4.7). Finally, *replica* is a copy of *subparent* which is equal to $corres^{-1}$; *subparent*; *corres* (e.g., $d' \mapsto a'$ and $e' \mapsto a'$ in Figure 4.7).

At this point, the reason we introduce a number of additional local parameters is to make models easier to read. For example, without introducing *des*, *subparent* and *replica*, *act*1 of Figure 4.8 can be replaced by

$$parent := parent \cup corres^{-1};\ (tcl(parent))^{-1}[\{obj\}] \lhd parent;\ corres \cup \{nobj \mapsto to\}$$

but we can see that the action becomes more difficult to read.

Additionally, there are two main constraints in this event. Firstly, the object being copied, *obj*, must exist and must not be the *root*. This is satisfied by *grd*1. Secondly, the destination, *to*, must exist and must not be the object being copied or its descendant

Event *copy* $\widehat{=}$
Any
obj, **to**, *des*, *objs*, *corres*, *nobjs*, *nobj*, *subparent*, *replica*
Where
$grd1 : obj \in objects \setminus \{root\}$
$grd2 : des \subseteq objects$
$grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd4 : to \in objects$
$grd5 : to \notin des \cup \{obj\}$
$grd6 : objs = des \cup \{obj\}$
$grd7 : nobjs \subseteq OBJECT \setminus objects$
$grd8 : corres \in objs \rightarrowtail\!\!\!\!\!\!\rightarrow nobjs$
$grd9 : nobj = corres(obj)$
$grd10 : subparent = des \lhd parent$
$grd11 : replica = corres^{-1};\ subparent;\ corres$
Then
$act1 : parent := parent \cup replica \cup \{nobj \mapsto to\}$
$act2 : objects := objects \cup nobjs$
End

FIGURE 4.8: A specification of copy event

(satisfied by *grd*5). Guard *grd*5 plays an important role to ensure that loops are not produced by this event.

Move event: This event is aimed at moving an existing object except *root* from one place to another place. Considering Figure 4.9, *a* is an object being moved from node *r* to node *c*. Node *c* will become a new parent of *a*. In Figure 4.10, an existing object named *obj* is moved to a new location named *to*. Parameter *des* represents the set of all descendants of *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$. In this case, the destination, *to*, must exist and not be *obj* or a descendant of *obj* (these constraints are specified as *grd*2 and *grd*5). These guards are necessary to guarantee that the move does not introduce a loop or unreachable objects. The *parent* function is updated so that *obj* has *to* as its parent.

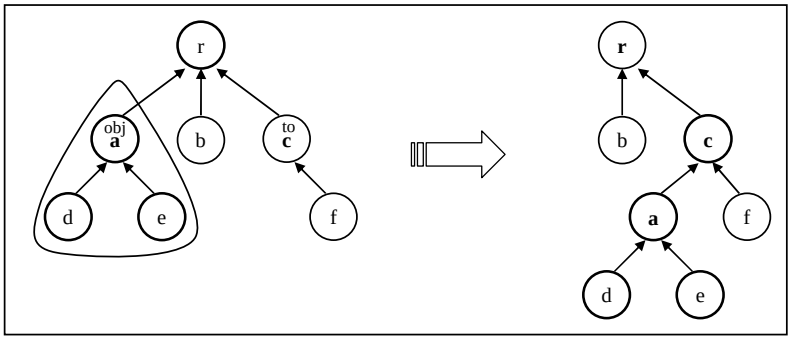


FIGURE 4.9: Diagram of moving a subtree rooted at *a* from *r* to *c*

Event *move* $\widehat{=}$
Any
obj, to, des
Where
$grd1 : obj \in objects \setminus \{root\}$
$grd2 : to \in objects$
$grd3 : des \subseteq objects$
$grd4 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd5 : to \notin des \cup \{obj\}$
Then
$act1 : parent(obj) := to$
End

FIGURE 4.10: A specification of move event

Delete event: This event is given in Figure 4.11. In this figure, *obj* is an object being deleted; *des* is a set of all *obj*'s descendants. Here $grd1$ states that *obj* must be an existing object except *root*. The object being deleted and all its descendants, *objs*, will be removed from *objects* by $act1$ and all related parent-entries are also removed by $act2$.

Event *delete* $\widehat{=}$
Any
$obj, des, objs$
Where
$grd1 : obj \in objects \setminus \{root\}$
$grd2 : des \subseteq objects$
$grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd4 : objs = des \cup \{obj\}$
Then
$act1 : objects := objects \setminus objs$
$act2 : parent := objs \lhd parent$
End

FIGURE 4.11: A specification of delete event

4.4 1^{st} Refinement: Files and Directories

In this refinement, objects are partitioned into files or directories. There are two machine variables introduced in this level, namely, *files* (a set of existing files) which is initialised to the empty set and *directories* (a set of existing directories) which is initialised to a set of *root*. Additionally, the *create* event of the abstraction is refined into events *crtfile*

(create file) and *mkdir* (make directory). Additional requirements for this level are given below.

Req2.1: The set of objects is partitioned into files and directories.

Req2.2: The root node is a directory.

Req2.3: The parent of each object must be a directory.

Figure 4.12 shows a list of machine variables, invariants formulated to satisfy the above requirements and initialised values of each variable. Considering the gluing invariant $inv2.4$, the abstract variable *objects* is entirely defined in terms of *files* and *directories*. As a result, it can be substituted by $files \cup directories$ and is no longer used in this level.

Variables
$files$, $directories$, $parent$
Invariants
$inv2.1 : files \subseteq objects$
$inv2.2 : directories \subseteq objects$
$inv2.3 : files \cap directories = \varnothing$
$inv2.4 : objects = files \cup directories$
$inv2.5 : root \in directories$
$inv2.6 : ran(parent) \subseteq directories$
Initialisation
$files := \varnothing$
$directories := \{root\}$
$parent := \varnothing$

FIGURE 4.12: Machine variables, invariants and initialisation of the first refinement

Because of the space constraint and the similarity of some events (such as creating a file and making directory), we chose two events (*crtfile* and *copy*) to illustrate a concrete model of this level.

Create-file event: This event (named *crtfile*), given in Figure 4.13, refines *create* of the previous abstraction. Additional details introduced in this refinement: (i) $grd2$, *in* must be a directory; and (ii) $act1$, the object must be added to the set *files* directly, instead of the set *objects* in the previous abstraction.

A refinement of Event *copy*: In this refinement, see Figure 4.14, additional details introduced in this event are: (i) $grd4$, the destination, *to*, must be a directory; (ii) $act2$, all correspondents of *objs* which are files must be added to the set *files*; and (iii) $act3$, all correspondents of *objs* which are directories must be added to the set *directories* as well. These two actions refine Action $act2$ of the previous abstraction (given in Figure 4.8).

Event *crtfile* **refines** *create* $\widehat{=}$
Any
obj, in
Where
$grd1 : obj \in OBJECT \setminus (files \cup directories)$
$grd2 : in \in directories$
Then
$act1 : files := files \cup \{obj\}$
$act2 : parent(obj) := in$
End

FIGURE 4.13: A specification of create-file event

Event *copy* **refines** *copy* $\widehat{=}$
Any
$obj, to, des, objs, corres, nobjs, nobj, subparent, replica$
Where
$grd1 : obj \in (files \cup directories) \setminus \{root\}$
$grd2 : des \subseteq (files \cup directories)$
$grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd4 : to \in directories$
$grd5 : to \notin des \cup \{obj\}$
$grd6 : objs = des \cup \{obj\}$
$grd7 : nobjs \subseteq OBJECT \setminus (files \cup directories)$
$grd8 : corres \in objs \rightarrowtail\!\!\!\!\!\!\rightarrow nobjs$
$grd9 : nobj = corres(obj)$
$grd10 : subparent = des \lhd parent$
$grd11 : replica = corres^{-1};\ subparent;\ corres$
Then
$act1 : parent := parent \cup replica \cup \{nobj \mapsto to\}$
$act2 : files := files \cup corres[objs \cap files]$
$act3 : directories := directories \cup corres[objs \cap directories]$
End

FIGURE 4.14: A first refinement of the copy event

The reason we have postponed files and directory to be introduced here is to make proof simpler. In the first level they are treated in the same way as objects which are nodes of the the tree structure. If we were to introduce them at the first step, proving the tree properties would be more difficult since files and directories are different. Namely, we would need to prove for both *crtfile* and *mkdir* events. At this level, we did not need to show that the *crtfile* and *mkdir* events preserve the tree properties, since their abstract event *create* has already been proved in the abstraction.

4.5 2^{nd} Refinement: File content

In this refinement, file contents and other related constraints are introduced together with five events – i.e. *r_open* (open an existing file for reading), *w_open* (open an existing file for writing), *read* (read the whole content of a file from the storage into a memory buffer), *write* (write the content of a file on the buffer back to the storage) and *close* (close an opened file). Note that we also introduce the *power_loss* event in this level, since the memory contents (read and write buffers) have been introduced. We postponed details of this event to be addressed in Section 4.11 where the fault-tolerance is outlined.

The requirements and constraints which are covered in this level:

Req3.1: Each file has content (which might be empty).

Req3.2: Each file must be opened before reading or writing.

Req3.3: A buffer of each opened file will be assigned once the file is opened and released when the file is closed.

Req3.4: All operations are disabled when the power is off.

Machine variables and invariants introduced in this refinement are listed in Figure 4.15.

Variables

...

$fcontent$, $power\_on$, r_opened_files, w_opened_files, $rbuffer$, $wbuffer$

Invariants

$inv3.1 : power\_on \in BOOL$

$inv3.2 : power\_on = TRUE \Rightarrow fcontent \in files \rightarrow CONTENT$

$inv3.3 : r\_opened\_files \subseteq files$

$inv3.4 : w\_opened\_files \subseteq files$

$inv3.5 : r\_opened\_files \cap w\_opened\_files = \varnothing$

$inv3.6 : rbuffer \in r\_opened\_files \rightarrow CONTENT$

$inv3.7 : wbuffer \in w\_opened\_files \rightarrow CONTENT$

FIGURE 4.15: Additional machine variables and invariants of the second refinement

In this refinement, the content of each file, *fcontent*, is defined as a total function mapped from each file to a content. The content is valid only when the power is on ($inv3.2$). Variable *r_opened_files* and *w_opened_files* are set of files which are opened for reading and writing respectively. The buffers of opened files, *rbuffer* (for reading) and *wbuffer* (for writing), are specified as a total function mapped from each opened file to a content.

The content is represented as an array of data items (DATA). In a context seen by this refined machine, the content is defined as a constant named *CONTENT*; and *DATA* is defined as a carrier set. We assume that the contents of each file are contiguous although it is specified as a partial function.

$$CONTENT = \mathbb{N} \nrightarrow DATA$$

Figure 4.16 given below represents an event *writefile*. This event aims to write the whole content of the given file named f on its buffer into the storage. The guard of the event ensures that the power must be on and the given file f must be opened for writing.

Event *writefile* $\widehat{=}$
  Any
    f
  Where
    $grd1 : power\_on = TRUE$
    $grd2 : f \in w\_opened\_files$
  Then
    $act1 : fcontent(f) := wbuffer(f)$
  End

FIGURE 4.16: A specification of file write event

4.6 3^{rd} Refinement: Permissions

In this level, requirements related to access permissions are introduced. The list of new requirements and constraints is given below.

Req4.1: Each object has an owner, a group-owner and a list of permissions.

Req4.2: Access to each object depends on its permissions.

Req4.3: Each user can be a member of one or more groups but mostly one primary group is assigned

Considering Figure 4.17, there are a number of machine variables introduced in this refinement. For example, *users*, a set of existing users; *groups*, a set of existing groups; *user_pgrp*, a primary group of each user; *user_grps*, user's groups; *obj_owner*, an owner of each object; and *obj_perms*, permissions of each object. Invariant *inv*4.5 states that a primary group of each user must be a group in which the user is a member. In a context seen by this machine, GROUP and USER are defined as a carrier set. PERMISSION, a set of permission types, is specified as a enumerated set which is equal to

$\{rbo, wbo, xbo, rbg, wbg, xbg, rbw, wbw, xbw\}$, where *rbo*: owner-read, *wbo*: owner-write, *xbo*: owner-execute, *gbo*: group-read, *wbg*: group-write, *xbg*: group-execute, *rbw*: world-read, *wbw*: world-write and *xbw*: world-execute.

Variables

...

users, *groups*, *user_pgrp*, *user_grps*, *obj_owner*, *obj_grp*, *obj_perms*

Invariants

*inv*4.1 : $users \subseteq USER$

*inv*4.2 : $groups \subseteq GROUP$

*inv*4.3 : $user\_pgrp \in users \rightarrow groups$

*inv*4.4 : $user\_grps \in users \leftrightarrow groups$

*inv*4.5 : $\forall u \cdot u \in users \Rightarrow user\_pgrp(u) \in user\_grps[\{u\}]$

*inv*4.6 : $obj\_owner \in (files \cup directories) \rightarrow users$

*inv*4.7 : $obj\_grp \in (files \cup directories) \rightarrow groups$

*inv*4.8 : $obj\_perms \in (files \cup directories) \leftrightarrow PERMISSION$

FIGURE 4.17: Additional machine variables and invariants of the third refinement

Figure 4.18 is an example of the *r_open* event, which is an extension of *r_open* in the previous abstraction. Italic lines represent the extending part that have added. Other part (not italic) inherited from the previous abstraction are shown here just for making the event more understandable. In this event, guards *grd*4 and *grd*5 state that user *usr* who issues this open request must exist and has a read-permission on the object *obj*.

Event *r_open* **extends** *r_open* $\widehat{=}$

Any

`f` *usr*

Where

`grd1` : `power_on` = `TRUE`

`grd2` : `f` $\in$ `files`

`grd3` : `f` $\notin$ `r_opened_files` $\cup$ `w_opened_files`

*grd*4 : $usr \in users$

*grd*5 : $f \mapsto usr \in RPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

Then

`act1` : `rbuffer(f)` := $\varnothing$

`act2` : `r_opened` := `r_opened_files` $\cup$ {`f`}

End

FIGURE 4.18: A specification of Event *r_open*

RPerm, which is used in the *r_open* event (shown in Figure 4.18), encodes the rules that determine whether a user has read permission for an object *obj*. It is defined in a context seen by this machine. The related part of the context defining *RPerm* is shown

in Figure 4.19. In the figure, *su* represents the super user (who has the right to manage every thing), defined as a USER constant. This function states that a user u has a permission to read an object o only if at least one of these criteria is satisfied:

(i) The user is the owner and has the owner-read permission (*rbo*).

(ii) The user is a member of the group to which the object belongs and has the group-read permission (*rbg*).

(iii) The world-read permission (*rbw*) is assigned to the object.

(iv) The user is the super user.

$$
\begin{array}{l}
o \mapsto u \in RPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps) \\
\quad \Leftrightarrow (\ (o \mapsto u \in obj\_owner \land o \mapsto rbo \in obj\_perms) \\
\qquad \lor \\
\qquad (obj\_grp(o) \in user\_grps[\{u\}] \land o \mapsto rbg \in obj\_perms) \\
\qquad \lor \\
\qquad (o \mapsto rbw \in obj\_perms) \\
\qquad \lor \\
\qquad (u = su)\ )
\end{array}
$$

FIGURE 4.19: A definition of read permission function

Other permission definitions (i.e., write and execute permission functions) which are not mentioned here are also specified in the same way.

Without providing this *RPerm* function, specifying Guard *grd*5 of Figure 4.18 would be more complicated in order to check whether the user has to the right to read the given file or not. Namely, this guard would be replaced by the specification given in Figure 4.20. Moreover, we would need to model like this for every event where the permission control is required. Instead, specifying as a separate permission function makes it reusable and easier to read.

$$
\begin{array}{l}
(\ (f \mapsto usr \in obj\_owner \land f \mapsto rbo \in obj\_perms) \\
\ \lor \\
\ (obj\_grp(f) \in user\_grps[\{usr\}] \land f \mapsto rbg \in obj\_perms) \\
\ \lor \\
\ (f \mapsto rbw \in obj\_perms) \\
\ \lor \\
\ (usr = su)\ )
\end{array}
$$

FIGURE 4.20: An alternative guard ensuring that *usr* has the read permission on f

4.7 4^{th} Refinement: Other missing properties

Other properties that have been missed or postponed at the previous abstract levels are explored in this level, for instance, creation date, last modification date and name. The event-extension feature is also used in this step to extend the model by adding these missing properties.

Here is an example of the *crt_file* event given in Figure 4.21. This figure shows some of the specification that have been extended. Parameter *nme* represents a name of the file being created. This name must not already exist in the given directory (*grd*8). Action *act*9 sets the creation date of the file being crated to be *nowdate*. We defined *nowdate* as a DATE constant in a context seen by this model. The last modification date is also set to be *nowdate*, while file size is initialised to be 0.

Event *crt_file* **extends** *crt_file* $\widehat{=}$
  Any *nme* **Where**
    $grd7 : nme \in NAME$
    $grd8 : nme \notin oname[parent^{-1}[\{indr\}]]$
  Then
    $act8 : oname(obj) := nme$
    $act9 : dateCreated(obj) := nowdate$
    $act10 : dateLastModified(obj) := nowdate$
    $act11 : file\_size(obj) := 0$
  End

FIGURE 4.21: An extended event *crtfile*

4.8 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 Figure 4.22 (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 of 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 of a file at the implementation level, we use fresh pages on the flash array rather than

over-writing the pages used for the previous version of the file. Additional details are explained in Section 4.9.

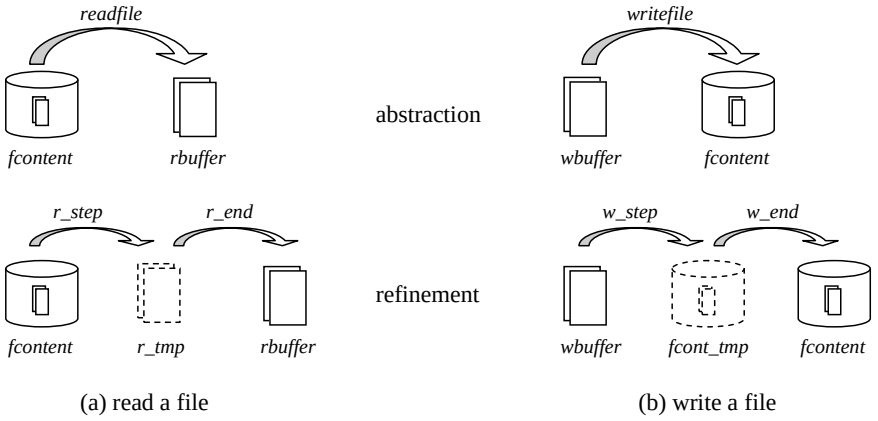


FIGURE 4.22: A diagram of refining events *readfile* and *writefile*

Note: 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 in Section 4.9. Full details of the specification can be found in Appendix A.

Other two structural refinements are (i) replacing an abstract file system by the flash specification which is outlined in Section 4.10 and (ii) decomposing a file system machine into two sub-machines to represent the file system layer and the flash interface layer. The details of the second one is provided in Section 4.13.

4.9 Decomposition of the file write event

Figure 4.23 (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 have been written completely). (The event decomposition we outline here follows the style of Butler and Yadav we have discussed in Section 3.5.) 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. 4.23 (b)), in the abstraction, the *writefileFail* event does nothing (i.e. *skip*). The content of file on the storage is not changed but all memory buffers are released.

Figure 4.24 shows machine invariants specified in this refinement step. Variable *fcont_tmp* represents the 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

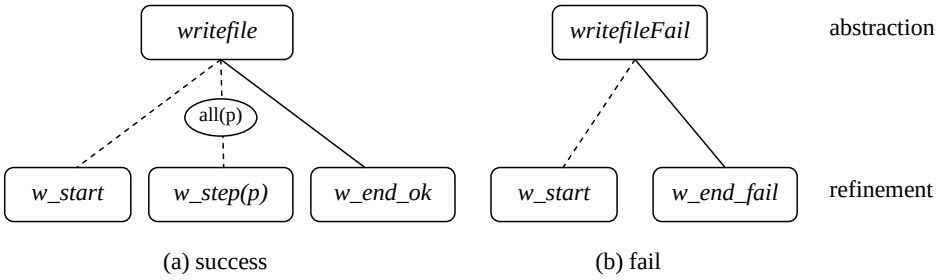


FIGURE 4.23: Refinement diagram of event *writefile*

to the shadow. No change is made to *fcontent* if writing of the given file fails at any point from the start to the end of writing. That means the content of that file will be the 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 *inv*6.3 states that for any file f which is in the writing state, the temporary contents of f will be a subset of or equal to the content on its writing buffer.

$$inv6.1 : writing \subseteq w\_opened\_files$$
$$inv6.2 : fcont\_tmp \in writing \rightarrow CONTENT$$
$$inv6.3 : \forall f \cdot f \in writing \Rightarrow fcont\_tmp(f) \subseteq wbuffer(f)$$

FIGURE 4.24: Machine invariants of the refinement

Figure 4.25 shows the refinement of event *writefile* when it is split into *w_start*, *w_step* and *w_end* (in cases of *success* and *fail*). In order to start writing (*w_start*), the given file must be opened for writing and not already in the writing state (see *grd2* and *grd3* 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 *grd2*). The page being written must be a page in the write buffer that has not already been written to the storage (see guards *grd5* and *grd6* of event *w_step*). Event *w_end_ok* is enabled when all pages have been written (*grd3*) 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 *grd3* 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 Figure 4.16). Namely, the gluing invariant, *inv6.3*, is used to show that *fcont_tmp(f)* is equal to *wbuffer(f)* when the guard of the *w_end_ok* event holds.

This model can deal with concurrent file read/write events. While a file is being written, another file might be read/written at the same time. Consider the file-write event, for example, where it is split into sub-events. Suppose file *f1* is requested to be written with three pages of contents, and file *f2* is also requested to be written with two pages at the

Event $w\_start \mathrel{\widehat{=}}$
  Any f **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in w\_opened\_files$
    $grd3 : f \notin writing$
  Then
    $act1 : writing := writing \cup \{f\}$
    $act2 : fcont\_tmp(f) := \varnothing$
  End
Event $w\_step \mathrel{\widehat{=}}$
  Any f, i, cnt **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in \mathbb{N}$
    $grd4 : cnt \in DATA$
    $grd5 : i \mapsto cnt \in wbuffer(f)$
    $grd6 : i \notin dom(fcont\_tmp(f))$
  Then
    $act1 : fcont\_tmp(f) := fcont\_tmp(f) \cup \{i \mapsto cnt\}$
  End
Event w_end_ok **refines** $writefile \mathrel{\widehat{=}}$
  Any f **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : dom(wbuffer(f)) = dom(fcont\_tmp(f))$
  Then
    $act1 : fcontent(f) := fcont\_tmp(f)$
    $act2 : writing := writing \setminus \{f\}$
    $act3 : fcont\_tmp := \{f\} \lhd fcont\_tmp$
    $act4 : file\_size(f) := card(fcont\_tmp(f))$
  End
Event $w\_end\_fail \mathrel{\widehat{=}}$
  Any f **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
  Then
    $act1 : writing := writing \setminus \{f\}$
    $act2 : fcont\_tmp := \{f\} \lhd fcont\_tmp$
  End

FIGURE 4.25: Decomposition of the *writefile* event

same time. Figure 4.26 shows two scenarios of concurrent file-write of both success and fail cases, where two file-write events have been performed in the same time.

Figure 4.26 shows that even when file *f1* has not been completely written we can start writing another file named *f2*. In addition, it is not necessary to complete writing page *p2* of file *f1* before start writing page *p3* of the same file. Moreover, although file *f1*

succeed	fail
$w\_start(f1)$	$w\_start(f1)$
$w\_step(f1, p1, cnt1)$	$w\_step(f1, p1, cnt1)$
$w\_start(f2)$	$w\_start(f2)$
$w\_step(f1, p3, cnt3)$	$w\_step(f1, p3, cnt3)$
$w\_step(f2, p1, cnt1)$	$w\_step(f2, p1, cnt1)$
$w\_step(f2, p2, cnt2)$	$w\_step(f2, p2, cnt2)$ fail
$w\_end\_ok(f2)$	$w\_end\_fail(f2)$
$w\_step(f1, p2, cnt2)$	$w\_step(f1, p2, cnt2)$
$w\_end\_ok(f1)$	$w\_end\_ok(f1)$

FIGURE 4.26: Scenarios of concurrent writing of two files

has been started first, it might be completed after the completion of writing file *f2*. If failures occur at any point (see Figure 4.26 (left) where writing page *p2* of file *f2* fails) *w_end_fail* will be reached instead of the *w_end_ok* event, since the failure will prevent *grd*3 of the *w_end_ok* event from becoming true.

4.10 Linking the Abstract File System to the Flash Interface Layer

This section outlines an initial model of the flash specification, which is based on the ONFI specification given in [52], 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 4.10.1 and then show a refinement of the file system layer when the flash specification is included.

4.10.1 Abstract Flash Interfaces Layer

An ONFI-based flash device is represented as 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. The ONFI structure means that flash pages are accessed via row addresses that consists of a LUN number, a block number within a LUN and a page number within a block. A flash device can be specified in Event-B as an array of pages which are identified by row addresses:

$$flash \in RowAddr \rightarrow PDATA$$

where *RowAddr* is specified as a carrier set representing all possible row addresses. In this step, we ignore the structure of the row address, since its components (i.e. LUN, block and page numbers) within a row address are not used/referenced in this level. The structure is postponed to be specified in another refinement step. We have found that

ignoring the structure of the row address makes the model of this refinement simpler, since row addresses are represented using a simple form as a carrier set. It would be more complex if we were to specify $RowAddr$ as $LUAddr \times BAddr \times PAddr$, where $LUAddr$: LUN addresses, $BAddr$: block addresses and $PAddr$: page addresses. An appropriate way for modelling the structure of the row address and details are discussed in Chapter 6 where further refinements focussing on the flash specification are explored.

$PDATA$ represents a page data within each page. However, the ONFI specification does not provide details of how data is stored in each page. In order to deal with faults, we have made an assumption that page data is composed of an actual data (to be stored), an object to which the data belongs, a logical page id or page index (in the view of file system) and a version number identifying the version of that page data. Figure 4.27 represents the structure of $PDATA$. We model each component of $PDATA$ as a projection function following the approach of Evans and Butler [53] that has already been discussed in Section 3.8. 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.

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

FIGURE 4.27: A structure of PDATA

We have tried an alternative way to specify the contents of the page data as machine variables. Namely, each property (i.e. $dataOfpage$, $objOfpage$, $pidxOfpage$ and $verOfpage$) is specified as a machine variable. We have found that this makes our model become more complex and difficult to manage. In addition, modifying the contents of a PDATA is made to the whole rather than some parts of the PDATA. For example, rewriting a page content of a file with a new content is done by writing the new content to another fresh page (rather than modifying the content at the old location) and then mark the old one as obsolete. Thus, specifying as machine variables that makes it be able to modify an individual part of page data is not necessary.

Moreover, we have tried another way to specify $flash$. Namely, instead of specifying $flash \in RowAddr \rightarrow PDATA$ as above, we could use curried functions to specify $flash$ by introducing two other type-constants, LUN and $BLOCK$, as

$$
\begin{array}{c}
BLOCK = PAddr \rightarrow PDATA \\
LUN = BAddr \rightarrow BLOCK
\end{array}
$$

and then define *flash* as

$$flash \in LUAddr \rightarrow LUN$$

This alternative choice shows that *flash* is a collection of LUNs instead of a collection of PDATAs, directly. We have done an experiment to compare these two approaches. We found that using curried function for this case study makes the model more difficult to specify and reason about. An example comparing both approaches is given in Figure 4.41 of Section 4.13.

4.10.2 Relating the File System Layer with the Flash Interface Layer

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 parts of the events *r_step* and *w_step* of the file system 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*, *fat_tmp* and *flash*. 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 within the flash. The corresponding row address represents the location (within the flash device) in which the content of that page is stored. Variable *flash* represents a flash content which is a collection of pages.

The properties mentioned above are described by the invariants given in Figure 4.28. Many invariants (e.g. *inv7.3*, *inv7.4*, *inv7.8*, *inv7.9* and *inv7.10*) are gluing invariants introduced to relate the abstract variables *fcontent* and *fcont_tmp* with the concrete variables *fat*, *fat_tmp* and *flash*. 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 obsolete. Invariants *inv7.8* and *inv7.10* relate the content of file with the actual content on the flash device. For instance, *inv*7.8 says that for any flash page with a version that equals 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 as defined by *content*. Invariant *inv7.10* ensures that the FAT table is formulated correctly from the right version of such pages.

Figure 4.29 illustrates how the file write of the abstract file system is replaced by the flash specification. The top diagram represents the abstract file write 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

$$
\begin{array}{l}
inv7.1 : fat \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr) \\
inv7.2 : fat\_tmp \in writing \rightarrow (\mathbb{N} \nrightarrow RowAddr) \\
inv7.3 : \forall f \cdot f \in files \Rightarrow dom(fat(f)) = dom(fcontent(f)) \\
inv7.4 : \forall f \cdot f \in files \wedge f \in writing \Rightarrow dom(fat\_tmp(f)) = dom(fcont\_tmp(f)) \\
\\
inv7.5 : flash \in RowAddr \rightarrow PDATA \\
inv7.6 : programmed\_pages \subseteq RowAddr \\
inv7.7 : obsolete\_pages \subseteq programmed\_pages \\
\\
inv7.8 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in files \\
\quad \wedge\ verOfpage(p) = curr\_version(objOfpage(p)) \wedge pidxOfpage(p) \neq 0 \\
\quad \Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcontent(objOfpage(p)) \\
\\
inv7.9 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in writing \\
\quad \wedge\ verOfpage(p) = writing\_version(objOfpage(p)) \wedge pidxOfpage(p) \neq 0 \\
\quad \Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcont\_tmp(objOfpage(p)) \\
\\
inv7.10 : \forall i, r, f, p \cdot f \in files \wedge r \in programmed\_pages \setminus obsolete\_pages \\
\quad \wedge\ p = flash(r) \wedge verOfpage(p) = curr\_version(f) \\
\quad \wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0 \\
\quad \Rightarrow\ i \mapsto r \in fat(f) \\
...
\end{array}
$$

FIGURE 4.28: Machine invariants for replacing the file system by the flash specification

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.

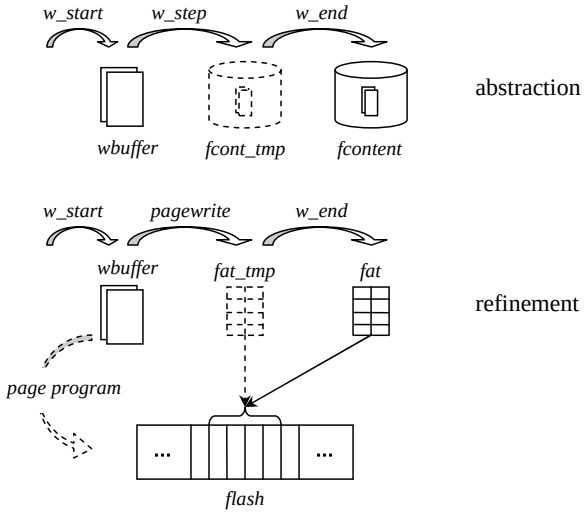


FIGURE 4.29: A diagram of mapping *writefile* to the flash specification

Figure 4.30 shows a simple example of data refinement in this level when the abstract

file system is replaced by the flash specification. The top represents an abstraction where the content of file $f1$ is represented by $fcontent(f1)$. The content of this file is composed of two pages of contents: $cnt1$ and $cnt2$. In the refinement (the bottom), the abstract *fcontent* is refined by concrete variables *fat* and *flash*. To get the content of page 1, since $fat(f1)(1) = r4$, $flash(r4) = pd4$ and $dataOfpage(pd4) = cnt1$, we then get $fcontent(f1)(1) = dataOfpage(flash(fat(f1)(1)))$.

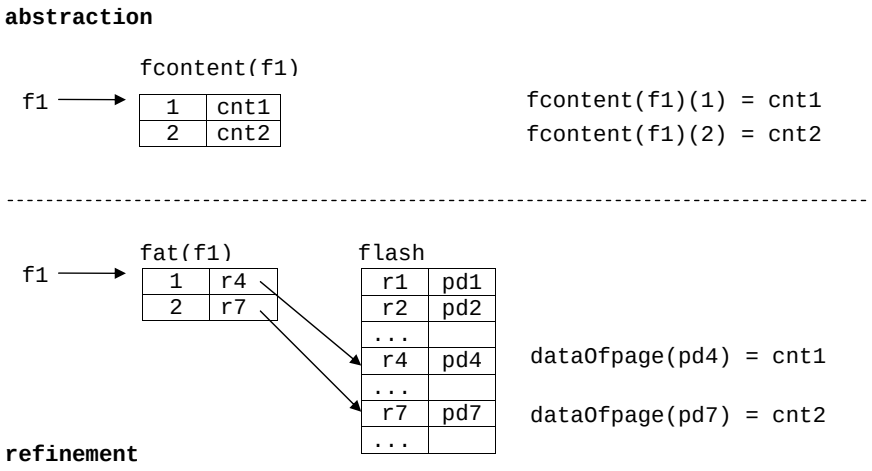


FIGURE 4.30: A diagram representing an example of data refinement where *fcontent* is replaced by *fat* and *flash*

Figure 4.31 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 (*grd7* and *grd8*) in order to write the content of page number i on the *wbuffer*. Parameter r represents a row address within the flash to which the content will be written. Based on the approach to modelling of records given in [53], we specify guards *grd9* - *grd13* to 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*.

Proof obligations generated by the Rodin tool help us to discover those gluing invariants (i.e. *inv*7.3, *inv*7.4, *inv*7.8 and *inv*7.9). For example, we have a generated PO typed *GRD* (guard strengthening proof obligation) to ensure the correct refinement of a guard (named *grd*6 of the *w_step* event, in Figure 4.25). Namely, we need to show that while the file f is in the writing state, $grd6 : i \notin dom(fcont\_tmp(f))$ (of the *w_step* event) of the previous abstraction is entailed by $grd6 : i \notin dom(fat\_tmp)$ of its refinement, given in Figure 4.31. This PO led us to Invariant *inv*7.4 saying that if file f is in the writing state, the domain of $fat\_tmp(f)$ equals the domain of $fcont\_tmp(f)$. Similarly for other gluing invariants, we used failing POs as guidelines.

Event *pagewrite* **refines** *w_step* $\widehat{=}$
  Any $f, i, cnt, r, pdata$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in \mathbb{N}$
    $grd4 : cnt \in DATA$
    $grd5 : i \mapsto cnt \in wbuffer(f)$
    $grd6 : i \notin dom(fat\_tmp(f))$
    $grd7 : r \in RowAddr$
    $grd8 : r \notin programmed\_pages$
    $grd9 : pdata \in PDATA$
    $grd10 : verOfpage(pdata) = writing\_version(f)$
    $grd11 : objOfpage(pdata) = f$
    $grd12 : lpidOfpage(pdata) = i$
    $grd13 : dataOfpage(pdata) = cnt$
  Then
    $act1 : fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
    $act2 : flash(r) := pdata$
    $act3 : programmed\_pages := programmed\_pages \cup \{r\}$
  End

FIGURE 4.31: The refinement of the *w_step* event

4.11 Dealing with faults

As previously mentioned in Section 4.1, our model tolerates faults that may occur at any point during the execution of file operations (e.g. reading and writing of files). Our fault model is based on the case that the system is able to reboot after failure. Our fault model deals with (i) power loss and (ii) failure to read or write a page of flash. Other failures such as fail-stop (that makes the system stops and is unable to reboot), Byzantine failure (processes fail by acting maliciously) [93] are not addressed. Our model covers both the file system software and the flash device. We assume that flash events (i.e. page read and page program) execute atomically. So that fault events are interleaved with non-fault events. In particular we assume that writing a page to flash either succeeds or fails in a detectable way.

To deal with faults, the use of a shadow disk and versioning has been employed in our model. This mechanism is a general standard which is widely used in file systems. Compared with other related work on verification of file system (where fault-tolerance were addressed), shadow disk and version numbers are also used in the work of Woodcock et al [118] and Kang et al [88].

In the case of power loss, all memory contents (such as buffers and the FAT table) are lost but the contents within the flash device remain. All file operations being executed are also aborted. The system needs to reformulate the correct FAT table from the flash

contents that have already been written prior to the power loss. Namely, the most recent version of such pages of file contents will be selected to formulate to FAT table when the power is on.

In the case of failure to read or program a page of flash, if reading/programming of any page (of any file) fails at the flash layer, the failure will be indicated to the file system layer. As a result, reading/writing of that file at the file system layer will be forced to abort. More details are discussed below.

Based on the characteristic of the flash device, modifying the content of a page must be done by writing the new content to another fresh page and then re-mapping the mapping table. Such pages of the file contents may have different versions. Suppose we want to write an existing file with a new content. The new content of that file will be written to another place (as a shadow), instead of modifying the content at the old location, with a newer version. The shadow content becomes the actual one if writing of that file has been completed. On the other hand, if any failure occurs during writing of a file, the previous valid version in the stage where the file was will be used. Some part of file contents (with the new version) that may be completely written will be ignored. Namely, the version number will be used to determine whether the page is the most recent version or not. The pages with the most recent version numbers will be selected to formulate the FAT table.

In our design, the file content is written to the flash device one page at a time. Writing of page data (or page program interface provided by the flash interface layer) is specified as an atomic event that can either succeed or fail. When all pages required have been written completely, page 0 (like the use of i-node of Unix file system [66]) will be written at the end in order to update the file description including the most recent version of that file. Thus, if any of pages required has not been written successfully, the page 0 will not be written (writing of that file will be aborted). That means the most recent version number of that file will not be updated. In mount stage, the system will know which one is the most recent one of the file content that will selected to formulate the correct FAT table, while other pages with invalid version numbers will be ignored.

In our development, we have introduced power loss and power on events to the model. The power loss event has the effect of releasing all memory contents, while the power on event has the effect of reconstructing the correct FAT table from the existing contents stored on the storage. These two events do nothing with the written data on the storage but the memory contents. Namely, no files or contents are changed or lost. Details of each event are discussed below.

Figure 4.32 shows the specification of the *power_loss* event which is introduced in the second refinement, where file contents and memory buffers are added. The *power_loss* event sets the *power_on* flag to be false ($act1$) and releases all memory contents, i.e. lists of files being opened for writing and reading ($act2$ and $act3$), and writing and reading

buffers ($act4$ and $act5$).

Event *power_loss* $\widehat{=}$
  When
    $grd1 : power\_on = TRUE$
  Then
    $act1 : power\_on := FALSE$
    $act2 : w\_opened\_files := \varnothing$
    $act3 : r\_opened\_files := \varnothing$
    $act4 : wbuffer := \varnothing$
    $act5 : rbuffer := \varnothing$
  End

FIGURE 4.32: The *power_loss* event of the second refinement

Similarly for other following refinement steps, memory contents that have been introduced in refinement steps (such as *fat* and *fat_tmp* in the seventh refinement) are also released by this event. In this refinement, we also have an invariant ($inv2.x$) saying that while the power is off all memory buffers are empty.

$$inv2.x : power\_on = FALSE \;\Rightarrow\; (w\_opened\_files = \varnothing \wedge r\_opened\_files = \varnothing \wedge wbuffer = \varnothing \wedge rbuffer = \varnothing)$$

Figure 4.33 shows the *power_on* event which is introduced in the second refinement. This event sets the *power_on* status to be TRUE. This makes all data and events available. We do not need to set all buffers to be empty, since the invariant specified above have guaranteed. Similarly, this event is refined gradually when new features/design details are added in other following refinement steps.

Event *power_on* $\widehat{=}$
  When
    $grd1 : power\_on = FALSE$
  Then
    $act1 : power\_on := TRUE$
  End

FIGURE 4.33: The *power_on* event of the second refinement

Figure 4.34 shows the *power_on* event of the seventh refinement where the flash specification has been introduced. When the power is on, the *power_on* event reconstructs the FAT table from the existing data that has been stored before the power loss. Parameter *ft* represents the FAT table being reconstructed. Guards $grd5$ and $grd6$ guarantee that only correct versions of file contents stored on the flash device are selected to construct the FAT table. As specified in $grd5$, the corresponding page selected to

formulate the table of content (ft) of each file must be the recent version of that file ($verOfpage(p) = curr\_version(f)$). Guard $grd3$ ensures that all pages of such files are read to formulate the FAT table. (We assume that the content of such a file starts at index 1).

Event $power\_on$ **refines** $power\_on \widehat{=}$
Any
ft
Where
$grd1 : power\_on = FALSE$
$grd2 : ft \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
$grd3 : \forall f \cdot f \in files \Rightarrow dom(ft(f)) = 1..file\_size(f)$
$grd4 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in dom(ft) \Rightarrow p \in ran(flash)$
$grd5 : \forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in files$
$\wedge\ p = flash(r) \wedge verOfpage(p) = curr\_version(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0$
$\Rightarrow\ i \mapsto r \in ft(f)$
$grd6 : \forall i, r, f, p \cdot f \in files \wedge r \in programmed\_pages \setminus obsolete\_pages$
$\wedge\ p = flash(r) \wedge i \mapsto r \in ft(f)$
$\Rightarrow\ (\ verOfpage(p) = curr\_version(f) \wedge$
$objOfpage(p) = f \wedge pidxOfpage(p) = i\ )$
Then
$act1 : power\_on := TRUE$
$act2 : fat := ft$
End

FIGURE 4.34: The $power\_on$ event of the seventh refinement

4.12 Modelling of the mount event

The *mount* event we specified here is aimed at mounting the contents within the storage device into the file system. Figure 4.35 shows the specification of this event which is specified in the first level. This event has an effect of adding a subtree (prt) rooted at x into an existing file system. This subtree represents the file system structure within the device being mounted. The set of objects (i.e. files and directories) within the device which is mounted will be added to the set of existing objects ($act1$), where $objs$ represents the objects to be mounted. The parent structure is also updated by $act2$.

The *mount* event has been refined gradually in refinement steps, based on features and design details which are introduced in each step. Figure 4.36 shows an extended part of the *mount* event when the file content is introduced in the second refinement. In this step, guards $grd13$, $grd14$ and action $act4$ are added. $fcnt$ represents the content of each file within the device which is mounted. fs represents a set of files within the

Event *mount* $\widehat{=}$
  Any $objs, prt, x, fcnt$ **Where**
    $grd1 : objs \subseteq OBJECT$
    $grd2 : objects \cap objs = \varnothing$
    $grd3 : x \in objs$
    $grd4 : prt \in objs \setminus \{x\} \rightarrow objs$
    $grd5 : \forall s \cdot (s \subseteq prt^{-1}[s] \Rightarrow s = \varnothing)$
    $grd6 : prt \cap parent = \varnothing$
  Then
    $act1 : objects := objects \cup objs$
    $act2 : parent := parent \cup prt \cup \{x \mapsto root\}$
  End

FIGURE 4.35: The mount event of the initial model

device. Since the power loss is also introduced in this level, the *mount* event is enabled only when the power is on ($grd14$). As mentioned in Section 4.5, $power\_on$ is specified as a BOOL variable representing the power status. All events are disabled if the power is off ($power\_on = FALSE$).

Event *mount* **extends** *mount* $\widehat{=}$
  Any $fcnt$ **Where**
    $grd13 : fcnt \in fs \rightarrow CONTENT$
    $grd14 : power\_on = TRUE$
  Then
    $act4 : fcontent := fcontent \cup fcnt$
  End

FIGURE 4.36: The *mount* event of the second refinement

Similarly, when the flash is mounted, only valid pages with the most recent version are selected to formulate the FAT table. Figure 4.37 shows the seventh refinement of the *mount* event when the flash specification has been introduced. (Because of the space constraint, we will show only an important part of the *mount* event.) Guards $grd25$ and $grd26$ ensure that all pages which are read to formulate the FAT table (ft) are valid pages with the right version. The actions of the event add all information into the existing file system. (Full details of this event can be found in Appendix A).

4.13 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,

Event *mount* **refines** *mount* $\widehat{=}$
Any
$objs, fs, ds, prt, x, fcnt, objown, objperms$
$objgrp, objname, cdate, mdate, fsize, ft, crv$
Where
...
$grd22 : crv \in objs \rightarrow VERNUM$
$grd23 : ft \in fs \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
$grd24 : \forall f \cdot f \in fs \Rightarrow dom(ft(f)) = dom(fcnt(f))$
$grd25 : \forall p \cdot p \in ran(flash) \wedge objOfpage(p) \in dom(ft)$
$\wedge\ verOfpage(p) = crv(objOfpage(p))$
$\Rightarrow$
$pidxOfpage(p) \mapsto dataOfpage(p) \in fcnt(objOfpage(p))$
$grd26 : \forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages$
$\wedge\ f \in fs \wedge p = flash(r) \wedge verOfpage(p) = crv(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i$
$\Rightarrow\ i \mapsto r \in ft(f)$
Then
$act1 : files := files \cup fs$
$act2 : directories := directories \cup ds$
$act3 : parent := parent \cup prt \cup \{x \mapsto root\}$
$act4 : fat := fat \cup ft$
$act5 : obj\_owner := obj\_owner \cup objown$
$act6 : obj\_perms := obj\_perms \cup objperms$
$act7 : obj\_grp := obj\_grp \cup objgrp$
$act8 : oname := oname \cup objname$
$act9 : dateCreated := dateCreated \cup cdate$
$act10 : dateLastModified := dateLastModified \cup mdate$
$act11 : file\_size := file\_size \cup fsize$
$act12 : current\_version := current\_version \cup crv$
End

FIGURE 4.37: Part of the *mount* event of the seventh refinement

further refinements of the flash interface layer can be explored separately. The machine decomposition we apply here follows the style of Butler described in [26] that we have already discussed in Section 3.6. Namely, machine variables and events are partitioned into sub-machines. Sub-machines interact with each other via synchronisation over shared parameterised events.

Figure 4.38 shows a diagram of machine decomposition illustrating the decomposition of the events *pagewrite* and *pageread*. The top layer represents part of the file system that consists of machine variables *fat*, *fat_tmp*, *wbuffer*, and so on. The bottom layer represents part of the flash interface containing machine variables: *flash*, *programmed_pages* and *obsolete_pages*. The ovals represent synchronisation over shared parameterised events between the sub-machines. In this case, both sub-machines in-

teract with each other by synchronising over the *page_write* and the *page_read* events.

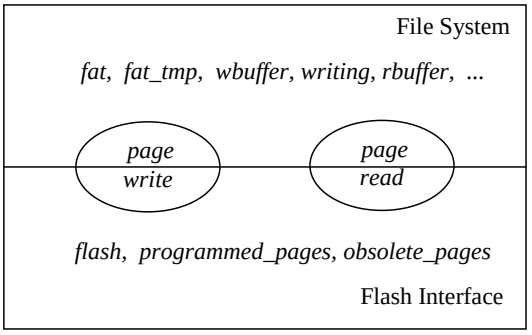


FIGURE 4.38: A machine-decomposition diagram focusing on events *page_read* and *page_write*

At this point, for example, we partition the *pagewrite* event given in Figure 4.31 following the approach of [26] (that we have already discussed in Section 3.8) and get a specification of the *page_program* event of the flash interface layer which is shown in Figure 4.39. We also get a specification of the *pagewrite* event of the file system layer given in Figure 4.40. Parameters r and *pdata* represent shared parameters which are used for an interaction between these two events.

Event $page\_program \mathrel{\widehat{=}}$
  Any $r, pdata$ **Where**
    $grd1 : r \in RowAddr$
    $grd2 : r \notin programmed\_pages$
    $grd3 : pdata \in PDATA$
  Then
    $act1 : flash(r) := pdata$
    $act2 : programmed\_pages := programmed\_pages \cup \{r\}$
  End

FIGURE 4.39: An abstract *page_program* of the flash interface layer

After decomposition, we get a machine specifying the flash interface layer which consists of two main events *page_program* and *page_read*. This machine can later be refined separately from the specification of the file system (in Chapter 6). We also get a machine specifying the file system with *pagewrite* and *pageread* plus the other events from earlier refinement such as *w_start* and *w_end*.

In Section 4.10.1, we discussed the alternative of using curried functions to model the flash structure. At this point, if we were to use curried function, the *page_program* event would be specified as the specification given in Figure 4.41. This event becomes more complicated, as we can see a number of parameters and guards are required for this

Event *pagewrite* $\widehat{=}$
  Any $f, i, cnt, r, pdata$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in \mathbb{N}$
    $grd4 : cnt \in DATA$
    $grd5 : i \mapsto cnt \in wbuffer(f)$
    $grd6 : i \notin dom(fat\_tmp(f))$
    $grd7 : r \in RowAddr$
    $grd8 : pdata \in PDATA$
    $grd9 : verOfpage(pdata) = writing\_version(f)$
    $grd10 : objOfpage(pdata) = f$
    $grd11 : lpidOfpage(pdata) = i$
    $grd12 : dataOfpage(pdata) = cnt$
  Then
    $act1 : fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
  End

FIGURE 4.40: Event *pagewrite* of the file system layer

event, compared with Figure 4.39. This would also make the *pagewrite* event (given in Figure 4.31) and the model more complex and difficult to manage and prove.

Event *page_program* $\widehat{=}$
  Any
    $lid, bid, pid, pcnt, old\_bk, old\_lun, new\_bk, new\_lun$
  Where
    $grd1 : lid \in LAddr$
    $grd2 : bid \in BAddr$
    $grd3 : pid \in PAddr$
    $grd4 : lid \mapsto old\_lun \in flash$
    $grd5 : bid \mapsto old\_bk \in old\_lun$
    $grd6 : pdata \in PDATA$
    $grd7 : new\_bk \in BLOCK$
    $grd8 : new\_lun \in LUN$
    $grd9 : new\_bk = old\_bk \lhd\!\!+ \{pid \mapsto pdata\}$
    $grd10 : new\_lun = old\_lun \lhd\!\!+ \{bid \mapsto new\_bk\}$
  Then
    $act1 : flash := flash \lhd\!\!+ \{lid \mapsto new\_lun\}$
  End

FIGURE 4.41: Event *page_program*, in case of using curried function

4.14 Proofs

The proof statistics, given in Table 4.1, show that 597 proof obligations were generated by the Rodin platform for all of the development outline in this chapter. 544 proof obligations (or 91%) were proved automatically while others were discharged interactively using the Rodin tool. MCH0 represents an initial model while MCH1 up to MCH7 represent refining machines in such refinement steps. CTX0 up to CTX3 represent contexts which are seen by those machines. (Note that proof statistics given here are slightly different from the proof statistics given in [44] because we have added additional events *mount*, *unmount*, *power_on* and *power_loss* in this development.) It can be seen that we have the high number of POs that were discharged interactively in MCH0 because proving tree properties is not easy, compared with other levels that have simpler properties. This is similar to the seventh refinement where we introduced the flash specification. This requires a number of gluing invariants that are not easy to prove automatically.

TABLE 4.1: Proof statistics

Machines/Contexts	Total POs	Automatic	Interactive
CTX0	10	8	2
CTX1	7	3	4
CTX2	0	0	0
CTX3	3	3	0
MCH0	45	30	15
MCH1	84	78	6
MCH2	51	51	0
MCH3	46	43	3
MCH4	43	42	1
MCH5	38	37	1
MCH6	42	41	1
MCH7	228	198	20
Overall	597	544 (91%)	53 (9%)

To make proof simpler, careful selection of invariants and machine theorems was important and eased the proof effort. For example, for the high-level requirements on the data structure, we introduced two tree properties: (i) no-loop and (ii) reachability. These properties are normally expressed using transitive closure. However, we identified simpler but sufficient formulations (*inv*1.3 and *inv*1.4 given in Figure 4.2) and expressed these as invariants. Proving that all events preserved these invariants was not too difficult since they did not involve transitive closure. The transitive closure formulations were expressed as machine theorems, and we showed that these followed from the existing invariants. We did not need to prove that the theorems were preserved by all machine events. This simplified the proof effort considerably.

In addition, to order to make interactive proofs easier, we introduced theorems that could

be reused for discharging several similar proof obligations. For instance, a theorem about *tree-join* was used to prove that the tree property holds for events *create*, *copy* and *move*. A theorem only needs to be proved once.

An important point is when we should introduce additional lemmas/theorems to help proofs. Based on our experience, we would like to suggest developers to introduce additional theorems if it is found that proofs of some POs are similar. Namely, they have similar goals and proof steps. Steps of proving those goals could be generalised and used to discharge similar POs. For example, proving the preservation of the no-loop property of the *copy* and *create* events is similar.

In proving that copy and create preserve the no-loop property, we had two similar goals given below. (To make it easier to follow, we named them as *GA* (for copy) and *GB* (for create).)

$$GA: \ \forall s \cdot s \subseteq (parent \cup replica \cup \{nobj \mapsto to\})^{-1}[s] \Rightarrow s = \varnothing \ ,$$

where *replica* represents a copy of the subtree being copied to node *to*; *nobj* represents the root node of the copy. Similar to the *create* event, we also have a similar goal given below (where $replica = \varnothing$, since there is only one node to be added).

$$GB: \ \forall s \cdot s \subseteq (parent \cup \{obj \mapsto indr\})^{-1}[s] \Rightarrow s = \varnothing \ ,$$

where *obj* is an object being created and *indr* is its parent. Proof of these two goals involved a huge number of proof steps. Several proof steps (such as instantiation and adding hypothesis) were discharged interactively. Other trivial proof steps (such as simplification rewrites) were discharged automatically.

At that point, we realised that proof steps required for *GA* and *GB* were quite similar. Hence, in our development, these proof steps were generalised as a theorem named *thm*5 (join theorem) given below.

$$\begin{aligned}
thm5 : \forall f, g, t, u, x, M, N \cdot \\
& N \subseteq OBJECT \\
& \land \ M \subseteq OBJECT \\
& \land \ N \cap M = \varnothing \\
& \land \ t \in M \\
& \land \ f \in M \setminus \{t\} \rightarrow M \\
& \land \ u \in N \\
& \land \ g \in N \setminus \{u\} \rightarrow N \\
& \land \ x \in M \\
& \land \ (\forall A \cdot A \subseteq f^{-1}[A] \Rightarrow A = \varnothing) \\
& \land \ (\forall B \cdot B \subseteq g^{-1}[B] \Rightarrow B = \varnothing) \\
& \land \ f \cup g \cup \{u \mapsto x\} \in (M \cup N) \setminus \{t\} \rightarrow M \cup N
\end{aligned}$$

$$\Rightarrow$$
$$(\forall\, C \cdot C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \Rightarrow C = \varnothing)$$

This theorem says that if tree f rooted at t has no-loops and g which is rooted at u has no-loops, then the join of g and f at node x also has no-loops. (M represents a set of all nodes of tree f while N represents a set of all nodes of tree g). This theorem was used to prove GA by providing $f = parent$, $g = replica$, $t = root$, $u = obj$, $x = to$, $M = objects$ and $N = nobjs$, where $nobjs$ represents a set of new nodes which are copies of all nodes of the subtree to be copied. Similarly for proof of GB, this theorem was instantiated by providing $f = parent$, $g = \varnothing$, $t = root$, $u = obj$, $x = indr$, $M = objects$ and $N = \{obj\}$.

We also saw this pattern of proof steps was similar to the *move* event, as we can see the similarity of the pattern of moving and copying a subtree illustrated in Figure 4.9 and Figure 4.7. (This is also similar to the *create* event, since we realised that the object being created is also a subtree that has only one node.) Namely, this theorem could be used for proving the preservation of the no-loop property of the *move* event as well.

To make it more general, *IsTree* given in Figure 4.42 could be introduced as a predicate ensuring that function p (parent function) on set S is a tree rooted at r. We could use this predicate to construct a tree theorem (named tree-join which is shown in Figure 4.44). This theorem can be used to prove that events copy, create and move preserve the tree properties (e.g. no-loop). To understand more about this theorem, Figure 4.43 is given to illustrate how tree-join theorem is formulated.

$$\begin{array}{l} IsTree(S, p, r) \;\Leftrightarrow\; (\\ \qquad r \in S \\ \qquad \wedge\;\; p \in S \setminus \{r\} \rightarrow S \\ \qquad \wedge\;\; \forall\, S \cdot S \subseteq p^{-1}[S] \Rightarrow S = \varnothing \\ \qquad \wedge\;\; S \setminus \{r\} \subseteq (tcl(p))^{-1}[\{r\}] \\ ) \end{array}$$

FIGURE 4.42: A predicate describing the tree property

The theorem given in Figure 4.44 states that if f is a tree rooted at r on M, g is a tree rooted at u on N, and M and N are disjoint then the join of g with node x on f is a tree.

Initially, when we were specifying all features in one level rather than layering them over several refinements, we had a lot of difficulty in identifying sufficient invariants. There were some proof obligations that could not be discharged, because of the difficulty of finding sufficient invariants. Because of this difficulty, we then chose a different way to specify our model by using a multi-levelled refinement approach. We found that the multi-level approach helped us to factor out the difficulty of modelling and, to identify the right invariants. At the earlier stage of work, while we were specifying everything

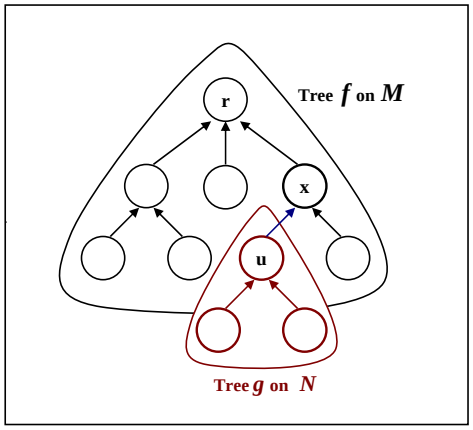


FIGURE 4.43: A diagram of tree join

$$
\begin{array}{l}
\mathit{TreeJoinThm} : \forall f, g, r, u, x, M, N \cdot \\
\quad M \subseteq \mathit{OBJECT} \land N \subseteq \mathit{OBJECT} \\
\quad \land \ \mathit{IsTree}(M, f, r) \\
\quad \land \ \mathit{IsTree}(N, g, u) \\
\quad \land \ M \cap N = \varnothing \\
\quad \land \ x \in M \\
\Rightarrow \\
\quad \mathit{IsTree}(M \cup N, f \cup g \cup \{u \mapsto x\}, r)
\end{array}
$$

FIGURE 4.44: A theorem of tree join

in one level, the model was complicated and we needed to identify a huge number of invariants. Some of them were related to some group of machine variables while some are independent from others. When we needed to modify some features, we needed to look through all invariants to find out which one should be modified. Sometimes, we missed some properties (e.g. forgot some invariants) and also needed to change the model. Compared with the multi-level approach, each level has an individual purpose or concentrates on just one feature (or just small set of features). This makes it easy to identify invariants focusing on just the features being addressed.

In addition, the multi-level approach also made our models easier to modify. For instance, when we wanted to modify the model that affects only one feature, we could go to the level where that feature was introduced, directly, and then modified what we wanted. In the case of horizontal refinement where the event-extension feature was used, the modification was propagated down automatically by the tool.

Having identified sufficient invariants using a multi-level approach we also experimented with collapsing MCH0 up to MCH4 to a single level. All invariants that had been discovered were merged into a single level. Figure 4.2 shows a comparison of the proof statistics of multi-level and single-level approaches, focussing on part of the horizontal refinement. The statistics show that the difference in automatic proof is not significant

between these two approaches. This may be because of the right invariants have already identified and proved by using the multi-level approach. Here we just collapsed them together, which is different from when we initially started with all features in a single level but straggled to find sufficient invariants.

TABLE 4.2: Proof statistics comparing multi-level with single-level approaches, focussing on horizontal refinement steps MCH0 up to MCH4

approach	Total POs	Automatic	Interactive
multi-level	260	235 (91%)	25 (9%)
single-level	239	208 (87%)	31 (13%)

From the table, we may see that the number of proof obligations of multi-level is higher than single-level. The reason is that some POs are required for proving to show the correct refinement of guards (*GRD*) and events (*SIM*). However, these were automatically discharged.

4.15 Conclusion and Assessment

In this chapter, we have outlined our development of a flash file system focussing on a tree-structured file system and basic file operations (such as create, open, read, write, delete, etc.), together with some experiments. The experiments which were carried out in this development are aimed at investigating which modelling styles and refinement approaches are suitable for our development. The purpose is to construct a model with clear and accurate formulation of the system properties and discharge of all proof obligations. To satisfy these, as discussed in the proof section, careful selection of invariants and machine theorems was important and eased the proof effort. For example, in the development of a file system, abstraction allows us to tackle difficulty properties (i.e. no-loop and reachability) in isolation of many other details. These properties are normally expressed using transitive closure. However, as discussed in the Proofs section, we selected simpler but sufficient formulations and exposed these as invariants.

In our development, we also have investigated, modelled and outlined the use of refinement in two different purposes. First, refinement was used in feature augmentation (or horizontal refinement) and the second was for structural refinement (or vertical refinement).

Feature augmentation was firstly used to construct a model of an abstract file system. Instead of specifying everything in one level that may increase proof difficulty, we decided to split the whole system features into sub-features. These sub-features were chosen to be introduced in refinement steps. Thus, each refinement step has its own purpose based on what features that have been introduced. As discussed in Section 4.14, we have found

that this approach helped us to identify sufficient invariants, and made the model easier to be constructed and modified. Namely, an incremental refinement (i.e. a small number of features/design details is added in each refinement step) makes the gap between each level of refinements smaller. The unique purpose of each refinement step and the smaller gap led to the easier identifying of invariants.

We have also found that the event-extension feature which is included in the new release of the Rodin platform (release 0.9.x and later) is very useful for horizontal refinements. As already discussed in Section 3.7 and [44], this feature makes models easier to be refined and modified.

Structural refinement was used for relating the abstract file system with the flash specification. Event-decomposition is a structural refinement on which we focused in Section 4.8. We have shown how the event-decomposition technique outlined in Section 3.5 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 4.9. We have found that the event-decomposition technique is very effective for breaking an atomic event. It can be applied to other work, that its 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.

When the flash specification has been introduced in the seventh refinement we have proceeded to another structural refinement to decompose the machine into two sub-machines (representing an abstract file system layer and a flash interface layer) in the following refinement step, using the machine-decomposition style of Butler [26]. These two layers interact with each other via the shared parameterised events. Based on this evidence, we believe that machine decomposition is useful for other developments with specification involving sub-systems that can be partitioned and refined separately. At the moment, Rodin did not provide any tool to decompose machines directly, we needed to decompose machines manually using the editor of the Rodin tool. After manually decomposition, we used the shared-event composition plug-in [111] to recompose the machines and show that the decomposition have been done correctly. Recently, a machine-decomposition tool [112] is available as a plug-in to the Rodin platform. This would be useful in the feature. The reason we decompose a machine is to enable further refinements focusing on the flash specification separately. (Details of further refinements are outlined in Chapter 6.)

In addition, we have shown that our model can deal with faults and concurrent file operations (e.g. read and write) in Section 4.8 and Section 4.9. While a file is being written, another file may be read or written at the same time, in an interleaved way. Failures might occur at any point during reading or writing of a file. As discussed in Section 4.8, use of a shadow and versioning, which is a general standard, was employed in our work. In order to write a file, the content of the given file is written to the shadow

one page at a time and this shadow becomes the actual content of the given file when all pages required have been completely written. This is the same style used in the work of Woodcock and Devies [118]. We also introduced version numbers to deal with this. If writing of a file with a new version failed, the previous version of that file will be used. All pages with the new version that have been partially written will be ignored. As mentioned earlier, the use of version numbers is also found in the work of Kang et al [88].

Note that a comparison with related work on specification and verification of flash file systems is provided in Chapter 7.

Chapter 5

Evolution of the File System Models and Proofs

5.1 Introduction

The aim of this chapter is to outline another version of the file system model, where the system requirements are partially changed. The changes mentioned are aimed at making the model deal with partial read/write of a file, which contrasts to the model outlined in Chapter 4 where the whole content of such file is read from or written to the storage. Another difference is the unbounded version numbers of file content. As specified in Chapter 4, the version number was bounded (i.e. 2-bit version is applied). The difficulty of the bounded version is the reusing of previous version numbers. Namely, before starting to write any page, we need to ensure that there are no valid pages with the version being reused. On the other hand, the unbounded version, the version number of each page will be increased every time it is rewritten without reusing the pervious version numbers. Since the life time of a flash device is limited by the limit erasure, a 32-bit number is large enough to be used for numbering the version of each page [62]. Therefore, the use of unbounded version numbers is reasonable.

In this chapter, we outline a revised model of the flash file system that aimed at covering those two requirements (partial read/write operations and unbounded version numbers) mentioned. Because the previous model and this revised model are quite similar, some features that have previously been specified in the former model can be reused. Parts of the model related to the new requirements are needed to be modified. We will outline the impact of these changes that affects the model. For example, which part of refinement chain are affected? How much specification can be reused? What is the difference in proof? Are the language and tool flexible enough to deal with this evolution?

Figure 5.1 shows a diagram of refinement chains representing an overview of our devel-

opment of a flash-based file system comparing the original model and the revised model. Two refinement chains are given in the figure. The original model is represented using suffix _*a* while _*b* represents the revised model. We also decomposed the seventh refinement machine into two sub-machines in the revised model, in order to represent the file system and the flash interface layers. Since the specifications of the abstract flash interface layer of both chains are the same, we used MCH_FL to represent the model of the flash interface layer for both modelling chains. This MCH_FL will be refined later in order to cover more details focussing on the flash specification (in Chapter 6).

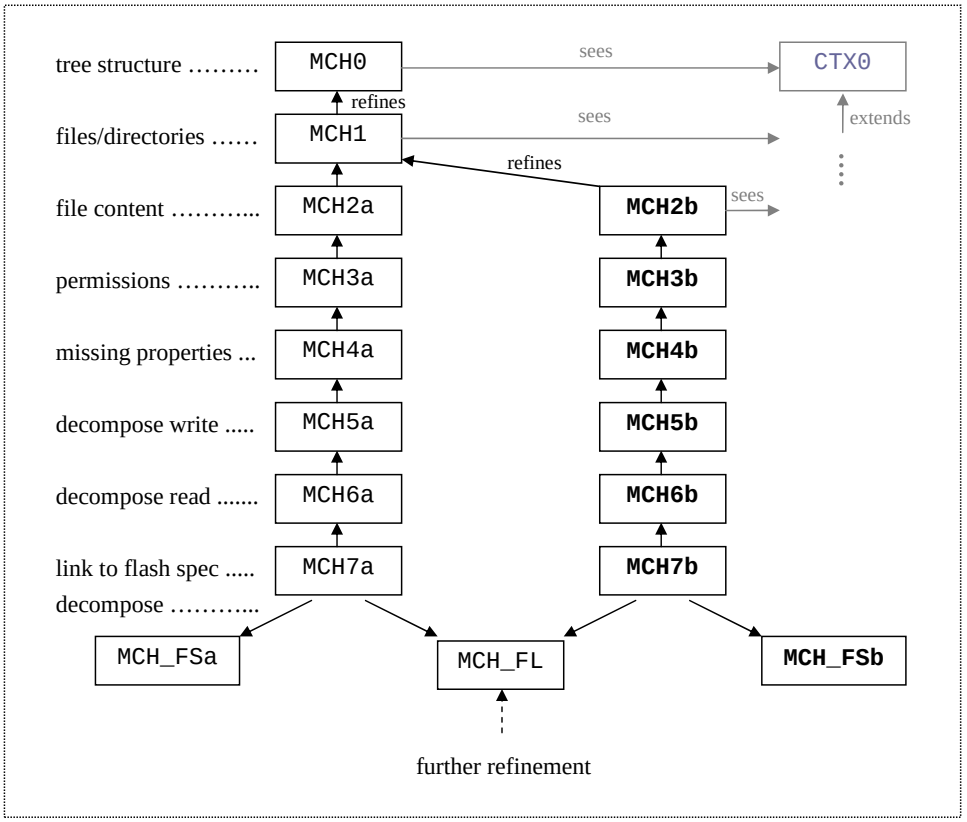


FIGURE 5.1: A diagram of refinement chains representing a flash file system

The revised model is also based on the tree-structure file system. The first and the second levels (MCH0 and MCH1) modelling the tree structure can also be reused in this development. No modifications are required in the first and the second levels. That means reproving tree properties are not required for the revised model, since they have already proved in the first level. To satisfy the new requirements, however, we need to modify the second refinement (MCH2) where file contents and related operations (e.g. read/write a file) are added. The modification which is made here is to meet the partial read/write requirement. From the third refinement down to the seventh refinement, the vertical refinement starting at the fifth refinement is the main part that needs to be modified. Details are discussed in Section 5.3.

Because of the similarity, this chapter outlines and discusses only important parts and

refinement steps that have been revised. The chapter begins with the second refinement where file contents have been introduced in Section 5.2. Section 5.3 gives modelling details where structural refinements have been explored to relate the file system layer to the flash interface layer. Finally, conclusion and assessment are given in Section 5.5.

5.2 2^{nd} Refinement: File content

Similar to the previous model given in Chapter 4, in this refinement, file contents and other related constraints are introduced together with five events: *r_open* (open an existing file for reading), *w_open* (open an existing file for writing), *read* (read the content of a file from the storage into a memory buffer), *write* (write the content of file on the buffer back to the storage) and *close* (close an opened file). Instead of reading/writing the whole content of file, partial read and write operations are allowed for this revised model.

The modification in this level does not affect existing invariants given in Figure 4.15. Only events *readfile* and *writefile* need to be changed to satisfy the partial read/write requirement. Figure 5.2 shows the revised version of the *readfile* and *writefile* events. The *readfile* event reads the content of the given file from the storage starting at the given offset with the length specified. Similarly, the *writefile* event is aimed at writing the content of the given file on the buffer into the storage starting at the given offset with the given length. (The length to be written must not be greater than the length of the content on the buffer.) Guards *grd*3 up to *grd*6 of the *readfile* event are added to restrict the scope of the contents to be read. Similarly for the *writefile* event, *grd*3 up to *grd*6 are added to ensure that the starting offset and length specified are valid. Guards *grd*7 and *grd*8 of the *writefile* event are aimed at specifying a mapping function (named *corresPos*) between logical addresses on the buffer and physical addresses of the file on the storage.

5.3 Vertical Refinement

Similar to the model outlined in Chapter 4, the purpose of vertical refinement here is to relate the abstract file system to the flash specification. The vertical refinement we explore in this section involves event and machine decomposition. The event 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. Three refinement steps are carried out: (i) decomposing the *writefile* event, (ii) decomposing the *readfile* event and (iii) decomposing the model into two sub-models.

Note: Instead of detailing the decomposition of both file read and file write which are

Event *readfile* $\widehat{=}$
  Any $f, offset, len$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in r\_opened\_files$
    $grd3 : offset \in dom(fcontent(f))$
    $grd4 : len \in \mathbb{N}$
    $grd5 : len \leq card(fcontent(f))$
    $grd6 : offset + len - 1 \in dom(fcontent(f))$
  Then
    $act1 : rbuffer(f) := (offset \mathinner{..} offset + len - 1) \lhd fcontent(f)$
  End

Event *writefile* $\widehat{=}$
  Any $f, offset, len, corresPos$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in w\_opened\_files$
    $grd3 : offset \in \mathbb{N}$
    $grd5 : len \in \mathbb{N}$
    $grd6 : len \leq card(wbuffer(f))$
    $grd7 : corresPos \in 0 \mathinner{..} len - 1 \rightarrowtail\!\!\!\!\!\rightarrow offset \mathinner{..} offset + len - 1$
    $grd8 : \forall p \cdot p \in dom(corresPos) \Rightarrow corresPos(p) = p + offset$
  Then
    $act1 : fcontent(f) := fcontent(f) \lhd\!\!\!\!- (corresPos^{-1};\ (0 \mathinner{..} len - 1 \lhd wbuffer(f)))$
  End

FIGURE 5.2: A specification of events *readfile* and *writefile*

similar, we present only file-write, which is more interesting in Section 5.3.1. Full details of the specification can be found in Appendix B.

Other two structural refinements are (i) replacing an abstract file system by the flash specification which is outlined in Section 5.3.2 and (ii) decomposing a file system model into two sub-models to represent the file system layer and the flash interface layer. The second one is detailed in Section 5.3.3.

5.3.1 A decomposition of the *writefile* event

An event refinement diagram given in Figure 4.23 can also be used to explain the decomposition of the *writefile* event. Namely, the *writefile* event is decomposed into three sub-events: *w_start* (start write), *w_step* (write a single) and *w_end* (end write, when all pages have been written completely). Event *w_end* refines *writefile* of the abstraction while *w_start* and *w_step* refine *skip*. Because of the requirement that has been changed, the specification in this refinement step is also changed.

Figure 5.3 shows machine invariants in this refinement step. Variable *fcont_tmp* repre-

sents temporary content of the file while it is in the writing state. As already discussed in Chapter 4, this variable behaves like a shadow content of the file being written. This shadow content becomes an actual content (*fcontent*) when all required pages have been written. 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 *inv*6.3 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.

$$
\begin{array}{l}
inv6.1 : writing \subseteq w\_opened\_files \\
inv6.2 : fcont\_tmp \in writing \rightarrow CONTENT \\
inv6.3 : \forall f \cdot f \in writing \Rightarrow fcont\_tmp(f) \subseteq wbuffer(f) \\
inv6.4 : writing\_offset \in writing \rightarrow \mathbb{N} \\
inv6.5 : writing\_len \in writing \rightarrow \mathbb{N} \\
inv6.6 : \forall f \cdot f \in writing \Rightarrow writing\_len(f) \leq card(wbuffer(f)) \\
inv6.7 : \forall f \cdot f \in writing \Rightarrow writing\_offset(f) \in dom(fcontent(f))
\end{array}
$$

FIGURE 5.3: Machine invariants of the refinement

Compared with the original model, two additional variables are introduced in this refinement: *writing_offset* and *writing_len*. The *writing_offset* variable is used to identify the starting position within the writing file to which the content will be written, while *writing_len* specifies the length of content to be written. Invariants *inv*6.1 up to *inv*6.3 are the same as specified in Chapter 4. The rest are additional invariants which are introduced to satisfy the partial write operation. For example, Invariant *inv*6.7 ensures that the offset used to start writing of any file must be in the valid domain.

Figure 5.4 shows the refinement of the *writefile* event when it is split into three phases. Consider the *w_start* event. Some changes have been made to this event. Namely, two additional parameters are added (i.e. *offset* and *len*). The given offset and length to be written must be valid (guarded by *grd*6 and *grd*7). The start event has an effect of putting the given file into the writing state and setting the scope of content to be written. 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 *grd5* and *grd6* of the *w_step* event). Event *w_end_ok* is reached when all pages required have been written (*grd7*) 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 starting at the offset specified.

Similar to the original model given in Chapter 4, Guard *grd7* 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 Figure 5.2). Namely, the gluing invariant, *inv6.3*, is used to show that *fcont_tmp(f)* is equal to *wbuffer(f)* when all guards of the

Event $w\_start \mathrel{\widehat{=}}$
  Any $f, offset, len$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in w\_opened\_files$
    $grd3 : f \notin writing$
    $grd4 : offset \in \mathbb{N}$
    $grd5 : len \in \mathbb{N}$
    $grd6 : len \leq card(wbuffer(f))$
    $grd7 : offset \in 0 \mathinner{..} file\_size(f)$
  Then
    $act1 : writing := writing \cup \{f\}$
    $act2 : fcont\_tmp(f) := \varnothing$
    $act3 : writing\_offset(f) := offset$
    $act4 : writing\_len(f) := len$
  End
Event $w\_step \mathrel{\widehat{=}}$
  Any $f, i, data$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in 0 \mathinner{..} (writing\_len(f) - 1)$
    $grd4 : data \in DATA$
    $grd5 : i \mapsto data \in wbuffer(f)$
    $grd6 : i \notin dom(fcont\_tmp(f))$
  Then
    $act1 : fcont\_tmp(f) := fcont\_tmp(f) \cup \{i \mapsto data\}$
  End
Event w_end_ok **refines** $writefile \mathrel{\widehat{=}}$
  Any $f, offset, len, corresPos, fsz$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : offset = writing\_offset(f)$
    $grd4 : len = writing\_len(f)$
    $grd5 : corresPos \in 0 \mathinner{..} len - 1 \rightarrowtail\!\!\!\!\!\!\rightarrow offset \mathinner{..} offset + len - 1$
    $grd6 : \forall p \cdot p \in dom(corresPos) \Rightarrow corresPos(p) = p + offset$
    $grd7 : dom(fcont\_tmp(f)) = 0 \mathinner{..} len - 1$
    $grd8 : fsz \in \{len + offset, file\_size(f)\}$
    $grd9 : fsz = len + offset \Leftrightarrow offset + len > file\_size(f)$
  Then
    $act1 : fcontent(f) := fcontent(f) \lhd\!\!\!\!- (corresPos^{-1}; fcont\_tmp(f))$
    $act2 : fcont\_tmp := \{f\} \lhd\!\!\!| fcont\_tmp$
    $act3 : file\_size(f) := fsz$
    $act4 : dateLastModified(f) := nowdate$
    $act5 : writing := writing \setminus \{f\}$
    $act6 : writing\_offset := \{f\} \lhd\!\!\!| writing\_offset$
    $act7 : writing\_len := \{f\} \lhd\!\!\!| writing\_len$
  End

FIGURE 5.4: Decomposition of the *writefile* event

w_end_ok event hold.

5.3.2 Linking the Abstract File System to the Flash Interface Layer

In this refinement step, flash properties are introduced together with variables used to relate the file system and the flash interface layers. Compared with the original version given in Chapter 4, there are no difference in specifying state variables. All machine variables can be reused in this revised model. Namely, variables *fcontent* and *fcont_tmp* of the file system layer are also 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 logical page-id of each file to its corresponding row address within the flash. The corresponding row address represents the location (in the flash) in which the content of that page is stored.

$$
\begin{array}{l}
\ldots \\
inv7.8 : \forall\, p \cdot p \in PDATA \\
\quad \wedge\ objOfpage(p) \in dom(fat) \\
\quad \wedge\ (\ \forall\, x \cdot x \in PDATA \wedge objOfpage(x) = objOfpage(p) \\
\qquad \wedge\ pidxOfpage(x) = pidxOfpage(p) \\
\qquad \Rightarrow verOfpage(x) < verOfpage(p)\ ) \\
\quad \Rightarrow \\
\quad pidxOfpage(p) \mapsto dataOfpage(p) \in fcontent(objOfpage(p)) \\
\\
inv7.9 : \forall\, p \cdot p \in PDATA \wedge objOfpage(p) \in dom(fat\_tmp) \\
\quad \wedge\ verOfpage(p) = writing\_version(objOfpage(p)) \\
\quad \Rightarrow \\
\quad pidxOfpage(p) \mapsto dataOfpage(p) \in wbuffer(objOfpage(p)) \\
\\
inv7.10 : \forall\, p \cdot p \in PDATA \wedge objOfpage(p) \in writing \\
\quad \Rightarrow \\
\quad writing\_version(objOfpage(p)) > most\_recent\_version(objOfpage(p)) \\
\\
inv7.11 : \forall\, i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in files \\
\quad \wedge\ p = flash(r) \wedge objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0 \\
\quad \wedge\ (\ \forall\, x \cdot x \in PDATA \wedge objOfpage(x) = f \\
\qquad \wedge\ pidxOfpage(x) = i \\
\qquad \Rightarrow\ verOfpage(x) < verOfpage(p)\ ) \\
\quad \Rightarrow\ i \mapsto r \in fat(f) \\
\ldots
\end{array}
$$

FIGURE 5.5: Machine invariants of replacing the file system by the flash specification

An important requirement affecting this refinement is the use of unbounded version numbers. Some modifications are required for the related events, i.e. create, read, write, etc. In addition, some invariants also need to be modified. Figure 5.5 shows some of machine invariants of the revised version. It is noted that Invariants *inv*7.1 up to *inv*7.7

are the same as specified in the original version. Here we show only some invariants that have been changed. Invariants *inv*7.8 and *inv*7.11 ensures that all pages that are used to generate the FAT table are the right versions (the most recent one) of such pages. For instance, *inv*7.8 says that for any page of the file within the FAT table where that page is the most recent version the given page-id, the data of that page will be the data of the given page-id of that file. Invariant *inv*7.9 ensures that the content of any page being written must be the content of the given file on the write-buffer. Finally, *inv*7.10 guarantees that the version of any page being written is greater than the most recent version of that page.

Figure 5.6 illustrates how the file write of the abstract file system is replaced by the flash specification. The top diagram represents the abstract file write 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* will be overridden by the *fat_tmp* when all required pages have been completely programmed into the flash device.

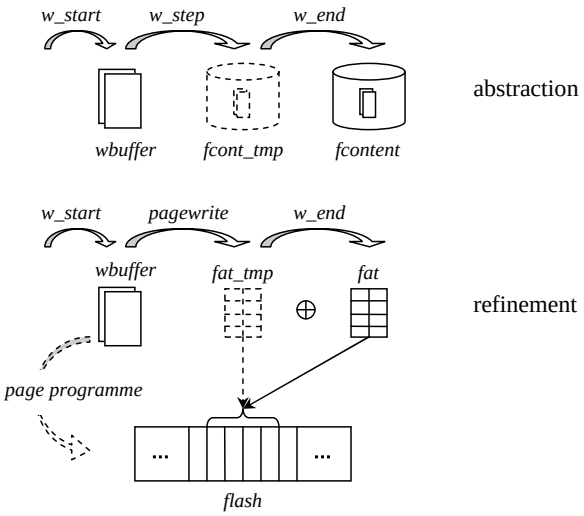


FIGURE 5.6: A diagram of mapping *writefile* to the flash specification

Figure 5.7 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 (*grd8* - *grd9*) to which the content of page number i on the *wbuffer* buffer is written. Parameter r represents a row address within the flash. *wv* represents the new version of the page being written. This version (*wv*) has been set by the start event which is equal to the latest version plus one. Guards *grd10-grd14* 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*. Action *act3* sets that row address as a programmed page.

Event *pagewrite* **refines** *w_step* $\widehat{=}$
  Any $f, i, data, r, pd, wv$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in 0 \mathinner{..} (writing\_len(f) - 1)$
    $grd4 : data \in DATA$
    $grd5 : i \mapsto data \in wbuffer(f)$
    $grd6 : i \notin dom(fat\_tmp(f))$
    $grd7 : wv = writing\_version(f)$
    $grd8 : r \in RowAddr$
    $grd9 : r \notin programmed\_pages$
    $grd10 : pd \in PDATA$
    $grd11 : objOfpage(pd) = f$
    $grd12 : pidxOfpage(pd) = i$
    $grd13 : verOfpage(pd) = wv$
    $grd14 : dataOfpage(pd) = data$
  Then
    $act1 : fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
    $act2 : flash(r) := pd$
    $act3 : programmed\_pages := programmed\_pages \cup \{r\}$
  End

FIGURE 5.7: The refinement of the *w_step* event

Figure 5.8 shows the refinement of the *w_end_ok* event. Guards *grd*1 to *grd*6 and actions *act*4 to *act*7 are similar to the previous abstraction given in Figure 5.4. Since variables *fcontent* and *fcont_tmp* are refined by *fat* and *fat_tmp*, Guard *grd*7 of this event is also changed (i.e. *fcont_tmp* is replaced by *fat_tmp*). This guard ensures that all pages required have been written. Local variable *toc* represents a table of contents which is a mapping function from each logical page id to the corresponding row address within the flash. The corresponding row address represents the location to which the content of that page id is programmed. Some changes are also made to the actions. For instance, Action *act*1 updates the table of content of the given file $fat(f)$. Action *act*1 releases the temporary FAT of the given file. Action *act*9 updates most recent version of the given file.

Figure 5.9 shows the *power_on* event of this evolution. This event is aimed at reconstructing the FAT table from existing data stored on the device. Similar to the original version (discussed in Section 4.11), guards *grd*5 and *grd*6 play an important role to ensure that page contents that have been read to construct the FAT table are valid pages with the most recent version.

Event w_end_ok **refines** $w\_end\_ok \mathrel{\widehat{=}}$
  Any $f, offset, len, toc, corresPos$ **Where**
    ...
    $grd7 : dom(fat\_tmp(f)) = 0 \mathinner{..} len - 1$
    $grd8 : toc \in \mathbb{N} \nrightarrow RowAddr$
    $grd9 : toc = corresPos^{-1};\ fat\_tmp(f)$
  Then
    $act1 : fat(f) := fat(f) \lhd\!\!\!- toc$
    $act2 : fat\_tmp := \{f\} \lhd fat\_tmp$
    ...
    $act8 : writing\_version := \{f\} \lhd writing\_version$
    $act9 : most\_recent\_version(f) := writing\_version(f)$
  End

FIGURE 5.8: The refinement of *w_end_ok* event

5.3.3 Machine Decomposition

The aim of this section is to outline the decomposition of the file system model that have been linked to the flash specification. In this step, we decompose the machine into a file system machine, modelling the file system layer, and a flash machine, modelling the flash interface layer, similar to what we have completed in Chapter 4. As a result, further refinements of the flash model can be explored separately. The machine decomposition we apply here also follows the style of Butler [26] outlined in Section 3.6.

Figure 5.10 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 variables *fat*, *fat_tmp*, *wbuffer*, and so on. The bottom layer represents the flash interface sub-machine containing variables named *flash, programmed_pages* and *obsolete_pages*. The ovals represent shared parameterised events used for synchronisation. In this case, both sub-machines interact with each other by synchronising over the *page_write* and the *page_read* events.

Figure 5.11 and Figure 5.12 show two parts of the *pagewrite* event (given in Figure 5.7) when it is partitioned following the approach of [26]. Figure 5.11 gives the specification representing *pagewrite* of the file system layer. Figure 5.12 represents the *page_program* (page program) interface provided by the flash interface layer. (We use different names to make the referencing of them easier.) Here we can see that the difference between the original model we presented in Chapter 4 and the revised model is the specification of the *pagewrite* event of the file system layer (Figure 5.11). The specification of the *page_program* of the flash interface model is the same as we obtained in the Chapter 4 (Figure 5.12). The requirements that have been changed affect only the file system layer. The specification of the flash interface layer can be reused from before. Parameters r and pd are shared parameters used for the interaction between these two events.

Event *power_on* **refines** *power_on* $\widehat{=}$
Any
ft
Where
$grd1 : power\_on = FLASE$
$grd2 : ft \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
$grd3 : \forall f \cdot f \in files \Rightarrow dom(ft(f)) = 1 \,..\, file\_size(f)$
$grd4 : \forall p \cdot p \in PDATA \wedge objOfpage(p) \in dom(ft) \Rightarrow p \in ran(flash)$
$grd5 : \forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in files$
$\wedge\ p = flash(r) \wedge objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0$
$\wedge\ (\ \forall x \cdot x \in PDATA \wedge objOfpage(x) = f$
$\wedge\ pidxOfpage(x) = i$
$\Rightarrow\ verOfpage(x) < verOfpage(p)\ )$
$\Rightarrow\ i \mapsto r \in ft(f)$
$grd6 : \forall i, r, f, p \cdot f \in files \wedge r \in programmed\_pages \setminus obsolete\_pages$
$\wedge\ p = flash(r) \wedge i \mapsto r \in ft(f)$
$\Rightarrow\ (\ verOfpage(p) < most\_recent\_version(f) \wedge$
$objOfpage(p) = f \wedge pidxOfpage(p) = i\ )$
Then
$act1 : power\_on := TRUE$
$act2 : fat := ft$
End

FIGURE 5.9: The *power_on* event of the seventh refinement

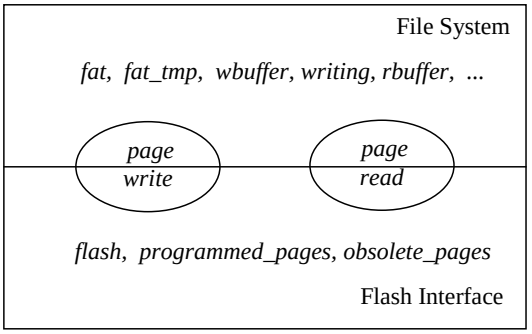


FIGURE 5.10: A machine-decomposition diagram focusing on events *page_read* and *page_write*

Similarly, after the decomposition is completed, we get a machine specifying the flash interface layer which consists of two main events *page_program* and *page_read*. This machine can later be refined separately from the specification of the file system. (Further refinements of the flash interface layer are given in Chapter 6.) We also get a machine specifying the file system with *pagewrite* and *pageread* plus the other events from earlier refinement such as *w_start* and *w_end*.

Event *pagewrite* $\widehat{=}$
  Any $f, i, data, r, pd, wv$ **Where**
    $grd1 : power\_on = TRUE$
    $grd2 : f \in writing$
    $grd3 : i \in 0 \mathinner{..} (writing\_len(f) - 1)$
    $grd4 : data \in DATA$
    $grd5 : i \mapsto data \in wbuffer(f)$
    $grd6 : i \notin dom(fat\_tmp(f))$
    $grd7 : wv = writing\_version(f)$
    $grd8 : r \in RowAddr$
    $grd10 : pd \in PDATA$
    $grd11 : objOfpage(pd) = f$
    $grd12 : pidxOfpage(pd) = i$
    $grd13 : verOfpage(pd) = wv$
    $grd14 : dataOfpage(pd) = data$
  Then
    $act1 : fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
  End

FIGURE 5.11: Event *pagewrite* of the file system layer

Event *page_program* $\widehat{=}$
  Any r, pd **Where**
    $grd8 : r \in RowAddr$
    $grd9 : r \notin programmed\_pages$
    $grd10 : pd \in PDATA$
  Then
    $act2 : flash(r) := pd$
    $act3 : programmed\_pages := programmed\_pages \cup \{r\}$
  End

FIGURE 5.12: An abstract *page_program* of the flash interface layer

5.4 Proofs

Table 5.1 shows the comparison of proof statistics between the original version of the file system and the revised version. To make it easier to compare, we also provided information given in brackets to represent proof statistics of the original version. Asterisks mean there is no difference between the original and the revised versions. In this development, 671 POs were generated automatically by the Rodin tool. 630 POs (94%) were proved automatically while the rest, 41 POs, were discharged interactively. As given in Table 5.1, it can be seen that some parts of modelling have not been affected (i.e. CTX0 up to MCH1 have no changes).

In this development of the revised version, we needed to reprove some POs interactively.

TABLE 5.1: Proof statistics – previous version in brackets

Machines/ Contexts	Total POs	Automatic	Reused Interactive	New Interactive
CTX0*	10	8	-	2
CTX1*	7	3	-	4
CTX2*	0	0	-	0
CTX3*	3	3	-	0
MCH0*	45	30	-	15
MCH1*	84	78	-	6
MCH2	56 (51)	56 (51)	0	0 (0)
MCH3	46	43	3	0 (3)
MCH4	43	42	1	0 (1)
MCH5	80 (38)	79 (37)	0	1 (1)
MCH6	81 (42)	80 (41)	0	1 (1)
MCH7	216 (228)	208 (198)	0	8 (20)
Overall	671 (597)	630, 94% (544, 91%)	4, 0.5%	37, 5.5% (53, 9%)

However, proving the same POs that have already been proved in the previous development is easier. Namely, we can reuse the proof tree of such PO that have already been discharged by copying it to discharge the same PO in the revised model. In this evolution, we have four proof trees that have been reused. (Details of reusing proof trees are discussed in Section 10.4) The number of POs of MCH5 and MCH6 are higher than the original model, since the partial write and read have been introduced. Namely, more constraints (e.g. offset and length to be read or written) need to be added. From the table, we can see that MCH7, where the unbounded version number is introduced, has a smaller number of interactive proofs, compared with the previous model. The smaller number of interactive proofs suggests that specifying using unbounded version numbers makes proof simpler. The version numbers will be reused, in the case of using bounded version numbers. This led to the difficulty of determining whether the page is the most recent version or not. Instead, in the case of using unbounded version numbers, we just increase the version number of such pages by 1 (if that page has been modified). Thus, the greatest version number of such a page is the most recent one of that page. This makes it easier to model and verify.

5.5 Conclusion and Assessment

We have presented the revised version of the file system model that have already been given in Chapter 4. The revision is based on the requirements that have been changed (i.e. partial read/write operation and unbounded version of the file contents). We have shown parts of the specification that were affected. The changes affected only parts where the file content is introduced and where the structural refinement has been taken place. We have found that the revised version where we used the unbounded version

number is easier to manage because using bounded version numbers makes model more complex than using unbounded version numbers. This is testified by proof statistics of the machine MCH7 given in Table 5.1.

In addition, we have found that parts of the feature augmentation have been slightly affected by the revision. As it can be seen that in earlier parts of the refinement chain (the first up to the fourth refinements), we needed to modify only the refinement step where the file content is introduced. By using the event-extension feature, this modification is propagated down automatically. Many parts of the original model given in Chapter 4 can be reused. The original contexts are completely reused without changes made. In addition, because the requirements that have been changed affected only part of the file system layer, the model representing the flash interface was not affected. That is, even if other requirements of the file system layer are changed – such as changing of the file system structure from the tree structure to the path-based structure – such a change will not affect the flash model.

The event-extension feature and the tools (e.g. modelling, refinement and proof) provided by Rodin are useful for this development. These make revising a model easy. Additionally, because of the facilities of tool and language, we can also model a system in different approaches (in different chains) in order to compare them. Since a machine can be refined by different machines, from the first level of a specification we may have several chains of refinement steps that can be used in comparison. This is also useful for studying and carrying out experiments in Event-B.

Chapter 6

Refinement of the Flash Interface Layer

6.1 Introduction

The purpose of this chapter is to outline a verified development of a flash interface layer (including refinements and proofs). As discussed in Chapter 4, after decomposition, the flash model will be refined separately by adding more details focusing on the flash specification in refinement steps. Further refinements mentioned are addressed in this chapter. For example, each LUN has at least one page register used for buffering data. Writing of a page is completed 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 at the given row address. Similarly for reading page data, the data will be first transferred to the page register before it is read into the memory buffer.

Additional events required for block reclamation are also explored in this chapter such as relocating a page and erasing a block. Reclamation involves selecting and erasing blocks in order to be reused for writing. In order to reclaim any block, the block should contain obsolete data. That means the number of free spaces will be increased when such a block is reclaimed. The candidate block (to be reclaimed) may have one or more pages with valid data. 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. Another constraint is that the number of erasures per block is limited (the number is dependent on its manufacturing), normally between 10,000 and 1,000,000 [62]. A block that fails to be erased becomes a bad block which can no longer use. The failures may be (i) the number of erasures has reached the erasure limit and (ii) the number of times that have been tried to erase the block have reached the limit

number.

Wear-levelling is a technique used for prolonging the life-time of the flash device. This technique involves selecting an appropriate block to be reclaimed in order to balance the number of erasures across the blocks within the flash chip. Namely, a block is worn-out (or is no longer to be used) when the number of erasures goes over the erasure limit. A summary of several wear-levelling techniques is given in [62] and [15]. We follow some of them in our development. Details are explained in each step of refinement.

In our development, concurrent page read/program is also covered. Reading/writing of pages can be performed simultaneously in an interleaved fashion. Each LUN has several page registers. While a page register is used for reading/writing of a flash page, another page register may be used for reading or writing of another flash page. Details are given in Section 6.2 where page registers are introduced. Concurrency is also applied to modelling of other processes such as the relocation process in Section 6.4 and the erasing process in Section 6.5.

Fault-tolerance is also addressed in our development. It can be seen in Section 6.4 and Section 6.5 where we outline the reclamation process that tolerates faults that may occur at any point during the block reclamation. The fault-tolerance of page-read and page-program operations has been dealt at the file system layer of Chapter 4. In this Chapter, the page-read and page-program events are also refined to deal with faults. In the case of faults (i.e. reading or programming a page fails), the status register of the corresponding LUN being performed will be set to indicate these faults. This makes the file system layer knows whether the reading/writing of a page succeeds or not.

This chapter starts with outlining further refinements that have been carried out in several refinement steps in Section 6.2 up to Section 6.6. The page register is introduced in the first refinement. The reclamation process is introduced in the second refinement and more details are added in the third and the fourth refinements. Finally, conclusions and assessment are given in Section 6.8.

6.2 1^{st} Refinement: Page Register

This refinement is based on the fact that two phases are required for the page read and the page program operations [52]. As stated in [52], in order to read from the flash array, the page data which is requested must be transferred to a page register before it is read off chip. In the case of the page program operation, data must be written to a page register before it is programmed into the flash array. To satisfy this, a page register is introduced as an intermediate buffer which is used as a temporary storage of a page data after it is read from or before it is programmed to the flash array.

In the ONFI architecture [52], page registers are intermediate buffers (RAM) within

LUNs. They are used for storing a page data after it is read from or before it is programmed to the flash array at a specified row address. Each LUN may have several page registers – depending on the number of interleaved operations supported per LUN. Thus, each page register is identified by a LUN address and an interleaved address within the LUN. We have compared two approaches for specifying page register addressing (PR): (i) cartesian product and (ii) projection functions. These two styles are mathematically equivalent.

The first approach (cartesian product) is specifying page register addressing as

$$PR = LUAddr \times IntAddr$$

where PR was specified as a constant representing set of page registers. Each page register is identified by a combination of a LUN address (LUAddr) and an interleaved address (IntAddr). Here $IntAddr$ is a set of interleaved addresses within a LUN (which is equal to 0..N-1 where N is the number of interleaved operations supported per LUN).

The second approach (projection function) is specifying PR as a carrier set in a context accompanied by two projections ($lidOfPR$, a LUN address to which each page register belongs and $intaOfPR$, an interleaved address of each page register), which are specified as constants. Axioms specifying these projections are given below.

$$lidOfPR \in PR \rightarrow LUAddr$$
$$intaOfPR \in PR \rightarrow IntAddr$$

In our experiment, we have found that these two approaches have no difference in proof (all POs were discharged automatically for both) but using projection function is more readable and easier to specify. Addressing an individual property within the cartesian product particularly when the product is composed of many entities like $RowAddr$ ($RowAddr = LUAddr \times BAddr \times PAddr$) is more complicated. For instance, addressing the $PAddr$ value of any row address r requires a nested projection (i.e. $prj2(prj2(r))$, where $prj2$ is the projection on the second element) which is more complicated. Therefore, we selected the projection function for our formulation as outlined in this report.

In our development, we classified the page registers into two different states based on what they are being used for. The first is $readingPR$, a set of page registers being used for reading. The second is $writingPR$, a set of page registers being used for writing. They were specified as machine variables given below.

$$readingPR \subseteq PR$$
$$writingPR \subseteq PR$$

where these sets are disjoint:

$$readingPR \cap writingPR = \varnothing$$

The corresponding row addresses to which the data within the page registers belong were formulated as

$$corresRowOfreadingPR \in readingPR \rightarrow RowAddr$$
$$corresRowOfwritingPR \in writingPR \rightarrowtail RowAddr$$
$$ran(corresRowOfreadingPR) \cap ran(corresRowOfwritingPR) = \varnothing$$

while the data within each page register was specified as a machine variable given below.

$$dataOfPR \in PR \rightarrow PDATA$$

We defined the corresponding row address of the page register being written as an injective because another write is not allowed to be performed on the same page being written. On the other hand, we specified the corresponding row address of the page register being read as a total function because multiple-reads can be performed on the same page.

In this refinement, the *page_read* and *page_prog* events were split into sub-events as given in Figure 6.1. For example, see (a) Page Read where the *page_read* event was decomposed into three steps (in order from left to right): (1) *pread_start*, selects an available page register within the LUN (to which the requested page belongs); (2) *read2reg*, transfers a page data into the selected page register; and (3) *pread_end*, reads data from the page register off chip. The *pread_end* event refines the abstract *page_read* while others refine *skip*.

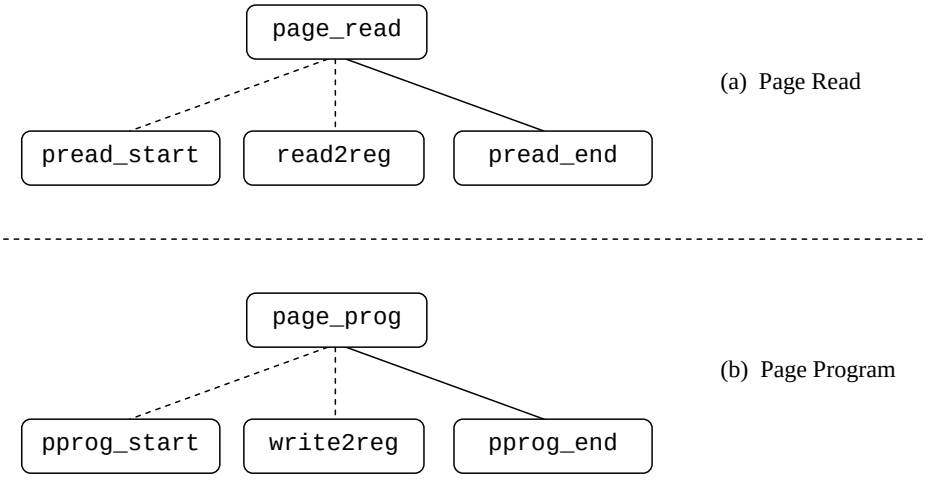
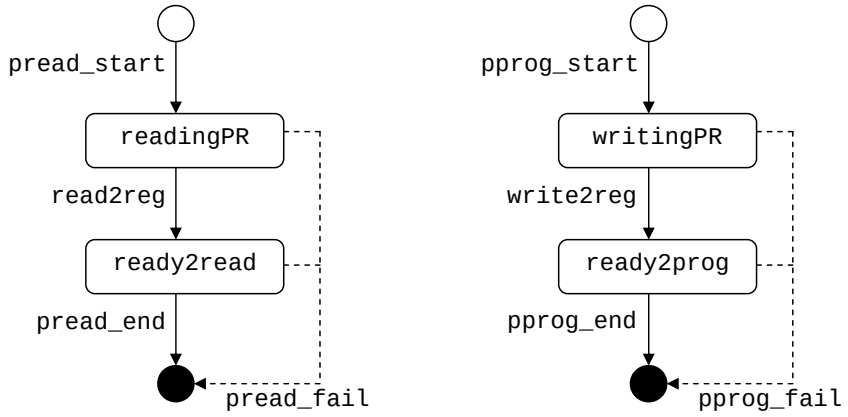


FIGURE 6.1: Event decomposition diagrams representing events *page_read* and *page_program*

In order to control the sequence of those sub-events, additional state variables are required. We introduced two state variables given below.



(a) Page Read (b) Page Program

FIGURE 6.2: State diagrams representing states of page registers which are used for reading and writing

$$ready2read \subseteq readingPR$$
$$ready2prog \subseteq writingPR$$

Variable *ready2read* represents a set of reading page registers with data that are ready for reading off chip. Variable *ready2prog* represents a set of writing page registers with data that are ready to be programmed into the flash array. Figure 6.2 shows state diagrams representing states of page registers, which are used for reading and writing. From the start event till the end event, failures may occur at any point. In the case of failures, those states will be reset. The *page_read* and *page_program* events are atomic events. They either completely succeed or fail. State transitions in Figure 6.2 correspond to the leaf events in Figure 6.1.

Figure 6.3 shows a refinement of the *page_read* event which was decomposed into three steps, as previously mentioned. In order to start reading (*pread_start* event) at row r, $grd1$ ensures that the given page must be valid (is already programmed and not obsolete). Parameter pr specifies an available PR within the LUN to which the given page belongs in order to be used for buffering (see $grd2$ to $grd4$). The actions put the page register pr into the reading state and set the corresponding row address of the page register to be the row address of the page being read, by $act1$ and $act2$ respectively. Event *read2reg* transfers the page content at the given row address into the corresponding page register. In this event, the page register must be in the reading state before transferring, and then it is set to be ready for reading (*ready2read*) once the content have been transferred to the page register. Event *pread_end* reads the content of the corresponding page register that have already been in the *ready2read* state off chip (see $grd1 - grd3$) and resets the page register (see $act1 - act3$).

Invariant $inv4$ given below is the gluing invariant we introduced in order to prove that the *pread_end* event is the correct refinement of the previous abstract event *page_read*.

Event *pread_start* $\widehat{=}$
  Any r, pr **Where**
    $grd1 : r \in programmed\_pages \setminus obsolete\_pages$
    $grd2 : pr \in PR$
    $grd3 : pr \notin readingPR \cup writingPR$
    $grd4 : lidOfRow(r) = lidOfPR(pr)$
  Then
    $act1 : readingPR := readingPR \cup \{pr\}$
    $act2 : corresRowOfreadingPR(pr) := r$
  End
Event *read2reg* $\widehat{=}$
  Any $r, pr, pdata$ **Where**
    $grd1 : pr \in dom(corresRowOfreadingPR)$
    $grd2 : r = corresRowOfreadingPR(pr)$
    $grd3 : pr \in readingPR$
    $grd4 : pr \notin ready2read$
    $grd5 : pdata \in PDATA$
    $grd6 : pdata = flash(r)$
  Then
    $act1 : dataOfPR(pr) := pdata$
    $act3 : ready2read := ready2read \cup \{pr\}$
  End
Event *pread_end* **refines** *page_read* $\widehat{=}$
  Any $r, pr, pdata$ **Where**
    $grd1 : pr \in ready2read$
    $grd2 : r = corresRowOfreadingPR(pr)$
    $grd3 : pdata = dataOfPR(pr)$
  Then
    $act1 : ready2read := ready2read \setminus \{pr\}$
    $act2 : readingPR := readingPR \setminus \{pr\}$
    $act3 : corresRowOfreadingPR := \{pr\} \lhd corresRowOfreadingPR$
  End

FIGURE 6.3: The first refinement of Event *page_read*

$$inv4 : \forall pr, r \cdot pr \in ready2read \wedge r \in programmed\_pages \wedge r = corresRowOfreadingPR(pr) \Rightarrow dataOfPR(pr) = flash(r)$$

This invariant says that if the corresponding page register (pr) of the page being read at row address r is in the *ready2read* state, then the content on the page register is equal to the page content of the flash at the given row r.

In the case of failures that may occur at any point from the *start* event to the last step of reading, the *fail* event is specified in Figure 6.4. This event is proved to refine *skip*. The page register being used for reading (see $grd1$) will be reset by $act1 - act3$. That means the page register will not be in the ready state that is valid to be read.

Event $pread\_fail \mathrel{\widehat{=}}$
  Any r, pr **Where**
    $grd1 : pr \in readingPR$
    $grd2 : r = corresRowOfreadingPR(pr)$
  Then
    $act1 : ready2read := ready2read \setminus \{pr\}$
    $act2 : readingPR := readingPR \setminus \{pr\}$
    $act3 : corresRowOfreadingPR := \{pr\} \lhd corresRowOfreadingPR$
  End

FIGURE 6.4: Event *pread_fail*

Our model allows concurrent interleaved reads and writes of different pages. Namely, while a page is being read into a page register, another page register may be used for reading or programming another page simultaneously. The number of interleaved events depends on the interleaved address supported per LUN. Considering the *pread_start* event (given in Figure 6.3), we can start reading another page if there is another page register available (see *grd*3 of the event).

6.3 2^{nd} Refinement: Events required for block reclamation

The purpose of the reclamation process is to select a block within a flash chip to be erased and reused. In order to erase a block, the given block must has no valid pages. If the given block contains valid pages, all valid pages must be relocated to another free block. Relocating a valid page is completed in two steps: (i) copy the valid content from the old location to a new location and (ii) mark the old location as obsolete at the end. These two steps are specified as events named *copy_a_page_to_new_loc* and *mark_old_page_obsolete*. Details will be explained later in this section.

Figure 6.5 shows some of machine invariants specified in this refinement. In order to relate an old location of any page that has been relocated to a new location, we introduced a translation function named *trans_func* which was specified as *inv*2.2. Variable *flash*2 represents part of the flash array that have been used for storing relocated pages. It is related to the *flash* in the view of the file system layer by the gluing invariant *inv*2.11. This invariant says that the content of page r (in the file system view) that have been relocated is equal to the content of page at the corresponding row of r (in the flash view).

The translation layer where the translation function is specified is a good design idea to deal with flash addressing. This mechanism avoids re-updating the FAT table when any valid page has been relocated to another location. When a flash page is requested to be read by the file system layer the translation layer has the responsibility of translating

Invariants

$inv2.1 : flash2 \in RowAddr \pfun PDATA$
$inv2.2 : trans\_func \in RowAddr \pfun RowAddr$
$inv2.3 : programmed\_pages2 \subseteq RowAddr$
$inv2.4 : dom(flash2) = programmed\_pages2$
$inv2.5 : dom(trans\_func) \subseteq programmed\_pages$
$inv2.6 : programmed\_pages2 = trans\_func[programmed\_pages]$
$inv2.7 : programmed\_pages \cap programmed\_pages2 = \varnothing$
$inv2.8 : obsolete\_pages2 \subseteq programmed\_pages \cup programmed\_pages2$
$inv2.9 : ran(trans\_func) \cap obsolete\_pages2 = \varnothing$
$inv2.10 : obsolete\_pages \subseteq obsolete\_pages2$
$inv2.11 : \forall r \cdot r \in dom(trans\_func) \Rightarrow flash(r) = (trans\_func;\ flash2)(r)$

FIGURE 6.5: Machine invariants of the second refinement

the requested page address to the corresponding location within the flash device. Then, the page data will be read and sent back to the file system layer. This translation table is designed to be stored in the memory. Although its content is lost in the case of power loss or sudden-reboot, all valid page contents still remain and are able to be used to re-formulate the correct FAT table at the mount stage which is dealt by the file system layer. Note that, at the moment, we did not model the translation layer separately. It is included as one feature of the flash interface layer we modelled in this chapter.

Figure 6.6 shows two additional events which were introduced in this refinement. Event *copy_a_page_to_new_loc* is aimed at copying the content of a valid page (*pdata*) from the old location (*old_r*) to the new location (*new_r*). Event *mark_old_page_obsolete* marks an old page at row *old_r* as obsolete. We introduced *programmed_pages2* as a set of pages that have been programmed during the relocation. It is specified as Invariant *inv*2.3 and *inv*2.6. We also introduced *obsolete_pages2* to represent an overall set of obsolete pages. As specified in Figure 6.5, it is a superset of obsolete pages of the previous abstraction.

However, the sequencing of events *copy_a_page_to_new_loc* and *mark_old_page_obsolete* was not addressed in this refinement. These events are independent and nondeterministically selected to be performed. In this step, we only prove that these events refine skip (i.e. executing them will conform to the previous abstraction). Sequencing control was postponed to the next refinement. The reason we modelled the feature in this way is to make our model simpler. Namely, introducing a small number of features raises a small set of POs required to be discharged. Additionally, it is easier to follow the model. At this point, developers can see that we can introduce a number of individual atomic events that are nondeterministically chosen to be performed at the level where they are introduced. After that, we can refine them later in a refinement step in order to control the sequence in which each event should be performed. To control the order of

Event $copy\_a\_page\_to\_new\_loc \mathrel{\widehat{=}}$
  Any $old\_r, new\_r, pdata$ **Where**
    $grd1 : old\_r \in programmed\_pages \setminus obsolete\_pages2$
    $grd2 : new\_r \in RowAddr \setminus (programmed\_pages \cup programmed\_pages2)$
    $grd3 : pdata = flash(old\_r)$
    $grd4 : old\_r \notin dom(trans\_func)$
  Then
    $act1 : flash2(new\_r) := pdata$
    $act2 : programmed\_pages2 := programmed\_pages2 \cup \{new\_r\}$
    $act3 : trans\_func(old\_r) := new\_r$
  End
Event $mark\_old\_page\_obsolete \mathrel{\widehat{=}}$
  Any old_r **Where**
    $grd1 : old\_r \in programmed\_pages$
    $grd2 : old\_r \notin obsolete\_pages2$
  Then
    $act1 : obsolete\_pages2 := obsolete\_pages2 \cup \{old\_r\}$
  End

FIGURE 6.6: Additional events required for reclamation process

events which are performed, additional state variables or flags are required and details are discussed in Section 6.4. Note that this technique only works when all steps of the process refine skip. In this development, the effect of relocation process is invisible to the file system.

6.4 3^{rd} Refinement: Ordering of Relocation Events

The purpose of this refinement is to control the sequence of the relocation events that has been postponed from the previous abstraction. The sequence of relocation events and related constraints are discussed below.

The block which is selected to be relocated may has some valid pages. Such a valid page within the selected block will be copied to another location and then mark the old one as obsolete. Relocating a block will end when all valid pages have been completely relocated. This process can be explained using a diagram given in Figure 6.7. This diagram shows an event-refinement diagram of the block relocation process. Note that we use dotted boxes to represent the abstract events *relocate_a_block* and *relocate_a_page* because there are no actual events specified in the abstract model. They are just abstract processes.

As illustrated in Figure 6.7, in order to relocate any block, three steps are required: (i) start relocating a block, (ii) relocate all valid pages within the block and (iii) end relocat-

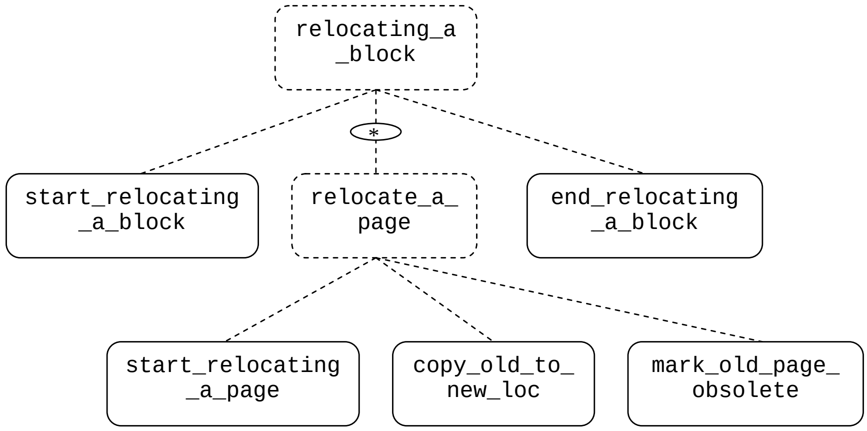


FIGURE 6.7: An event-refinement diagram representing the block relocation process

ing a block. Relocating a valid page is also completed in three steps: (i) start relocating a page, (ii) copy the valid content from the old location to the new location and (iii) mark the old location as obsolete at the end. Relocating a valid page is completed when the content of the given page has been copied to the new location and the old location has been marked as obsolete. Relocating a page will be repeated until all valid pages (within the block being relocated) have been relocated. Once the relocating block has no valid pages, the relocating process has been completed. This block becomes obsolete and is a candidate block that may be selected to be erased in the next process. As given in [86], erasing a block is not necessary to be performed once it has been completely relocated. That is, easing an obsolete block might be performed in background when the system is in the idle state or when free spaces are required. Details and refinement of erasing a block are discussed in Section 6.5.

Figure 6.8 shows machine invariants of this level. Two state variables were introduced. First, *relocating_blocks* represents a set of blocks that are in the relocating state. Second, *relocating_pages* represents a function relating the old location of each page being relocated to the new location, to which the content of this page is copied. Invariant *inv*3.3 says that all pages being relocated must be valid. Invariant *inv*3.4 says that all pages within the blocks being relocated are not allowed for reading and writing.

$$
\begin{aligned}
&inv3.1 : relocating\_blocks \subseteq BLOCK \\
&inv3.2 : relocating\_pages \in RowAddr \rightarrowtail\!\!\!\!\!\!+\; RowAddr \\
&inv3.3 : dom(relocating\_pages) \subseteq (programmed\_pages \setminus obsolete\_pages2) \\
&inv3.4 : \forall\, b, r \cdot b \in relocating\_blocks \wedge BlkOfRow(r) = b \\
&\qquad\quad \Rightarrow \\
&\qquad\quad r \notin ran(corresRowOfreadingPR) \wedge r \notin ran(corresRowOfwritingPR)
\end{aligned}
$$

FIGURE 6.8: Machine invariants of the third refinement

Figure 6.9 presents a refinement of *copy_a_page_to_new_loc* and *mark_old_page_obsolete*

events with ordering constraints added. Some changes were made to these events. For example, in order to copy a page from old_r to another location (new_r), $old\_r \mapsto new\_r$ must be in the relocating state (see $grd1$). Guard $grd2$ ensures that new location (new_r) must be free. In the second step, in order mark the old location to be obsolete, $old\_r \mapsto new\_r$ must be in the relocating state and new_r have already been programmed. Event $start\_relocating\_a\_page$ is new event refining skip. In order to start relocating a valid page (at row old_r), the given page must belong to the block being in the relocating state ($grd2$); and the new location (new_r) to which it is moved is free ($grd3$). If these two locations have not already been added to the relocating state, this event has an effect of setting them into the relocating state

Event $start\_relocating\_a\_page \mathrel{\widehat{=}}$
  Any old_r, new_r **Where**
    $grd1 : old\_r \in programmed\_pages \setminus obsolete\_pages2$
    $grd2 : BlkOfRow(old\_r) \in relocating\_blocks$
    $grd3 : new\_r \in RowAddr \setminus (progrmmed\_pages \cup programmed\_pages2)$
    $grd4 : old\_r \notin dom(relocating\_pages)$
    $grd5 : new\_r \notin ran(relocating\_pages)$
  Then
    $act1 : relocating\_pages := relocating\_pages \cup \{old\_r \mapsto new\_r\}$
  End
Event $copy\_a\_page\_to\_new\_loc$ **refines** $copy\_a\_page\_to\_new\_loc \mathrel{\widehat{=}}$
  Any $old\_r, new\_r, pdata$ **Where**
    $grd1 : old\_r \mapsto new\_r \in relocating\_pages$
    $grd2 : new\_r \notin programmed\_pages2$
    $grd3 : pdata = flash(old\_r)$
  Then
    $act1 : flash2(new\_r) := pdata$
    $act2 : programmed\_pages2 := programmed\_pages2 \cup \{new\_r\}$
    $act3 : trans\_func(old\_r) := new\_r$
  End
Event $mark\_old\_page\_obsolete$ **refines** $mark\_old\_page\_obsolete \mathrel{\widehat{=}}$
  Any old_r, new_r **Where**
    $grd1 : old\_r \mapsto new\_r \in relocating\_pages$
    $grd2 : new\_r \in programmed\_pages2$
  Then
    $act1 : obsolete\_pages2 := obsolete\_pages2 \cup \{old\_r\}$
  End

FIGURE 6.9: A refinement of page relocation

Note that our model can also deal with concurrent block relocation. Namely, while any block is in the relocating state, another candidate block can be relocated in the same time. Each sub-event of the relocation process is performed in an interleaved fashion. For example, while relocating a valid page of some blocks, another valid page of another

block can also relocated simultaneously.

Failures may occur at any point between the start and the end of relocating a block. The first case is failing to write the valid content to new location. This case does not pose any data inconsistency. That is, the content at the old location remains valid to be used while the new location is invalid to be used. Formally, it simply prevents further relocation steps and relocation always maintains consistency. The second is failing to mark the old one to be obsolete. In this case, two valid pages with the same content are stored in the flash array (at the old and the new locations). When the flash is remounted, only one valid page is firstly read and chosen to formulate the correct FAT table while another is marked as obsolete. This is not a problem because both locations have exactly the same content. Choosing either one of them to formulate the FAT table does not matter.

6.5 4^{th} Refinement: Refinement of Erasing a Block

The purpose of this section is to concentrate on the erasing process (or reclamation process) and outline what constraints we have addressed. In this refinement, the *block_erase* event is split into sub-events that can be performed in an interleaved fashion. Namely, our model presented here also deal with concurrent erase events.

In this refinement, several types of blocks were specified in order to classify and control the sequence of reclamation process. Figure 6.10 shows machine invariants specifying additional variables which were introduced.

candidate_blocks is a list of candidate blocks, which are allowed to be selected for reclamation.

relocating_blocks is set of blocks being relocated. It is a subset of candidate blocks.

obsolete_blocks is a set of blocks that have no valid pages (or the pages in use). They are candidate blocks that are ready to be erased in the reclamation process.

erasing_blocks is a set of obsolete blocks in the erasing state.

bad_blocks is a list of blocks that are no longer to be used. For instance, the block that fails to be erased will be marked as a bad block.

num_erased represents the number of times that each block has been erased. Each block can be erased within the maximum number allowed (the limited number depends on the manufacturing).

In our development, we decided to record the number of times that the blocks have been erased within each block in order to be used for the wear-levelling technique. The point

$$
\begin{aligned}
&inv4.1 : candidate\_blocks \subseteq BLOCK \\
&inv4.2 : relocating\_blocks \subseteq candidate\_blocks \\
&inv4.3 : obsolete\_blocks \subseteq BLOCK \\
&inv4.4 : obsolete\_blocks \cap relocating\_blocks = \varnothing \\
&inv4.5 : \forall r, b \cdot r \in programmed\_pages \land b \in obsolete\_blocks \\
&\qquad \land BlkOfRow(r) = b \;\Rightarrow\; r \in obsolete\_pages2 \\
&inv4.6 : erasing\_blocks \subseteq obsolete\_blocks \\
&inv4.7 : \forall b, r \cdot b \in obsolete\_blocks \land r \in RowAddr \land BlkOfRow(r) = b \\
&\qquad \Rightarrow \\
&\qquad (\, r \notin ran(corresRowOfreadingPR) \land r \notin ran(corresRowOfwritingPR) \,) \\
&inv4.8 : num\_erased \in BLOCK \rightarrow \mathbb{N} \\
&inv4.9 : invalid\_num\_erased\_blocks \subseteq BLOCK \\
&inv4.10 : restoring\_num\_erased \subseteq invalid\_num\_erased\_blocks \\
&inv4.11 : tmp\_num\_erased \in RowAddr \nrightarrow \mathbb{N} \\
&inv4.12 : corresBlkOftmpErased \in dom(tmp\_num\_erased) \rightarrow BLOCK \\
&inv4.13 : bad\_blocks \subseteq BLOCK \\
&inv4.14 : bad\_blocks \cap candidate\_blocks = \varnothing
\end{aligned}
$$

FIGURE 6.10: Machine invariants of the fourth refinement

is how to maintain the number of erasures when the block is erased. How to deal with failures that may occur during the erasing process. The idea of Marshall and Manning given in [62] is chosen as our solution. Namely, prior to erasing any block, the current number of erasures of that block must be copied to somewhere else. When the erasing step has been completed the number of erasures will be restored at the end. (Details are explained later in this section.) Here are additional machine variables which are introduced to deal with this.

invalid_num_erased_blocks represents the (erased) blocks with an invalid number of erasures. This kind of blocks becomes valid when the valid number of erasures has been restored.

restoring_num_erased represents a set of blocks that are in the state of restoring the number of erasures.

tmp_num_erased is used for recording the number of erasures of the blocks being erased. The number of erasures will be temporarily stored in another block that its state is not *erasing* or *relocating*.

corresBlkOftmpErased is a mapping function representing the associate block to which the temporary number of erasures belongs.

Figure 6.11 shows an event refinement diagram of the block-erase event which is composed of four sub-events: *start_erase_a_block*, *erase_a_block*, *start_restore_num_erased* and *restore_num_erased*. The *erase_a_block* event refines the previous *block_erase* event

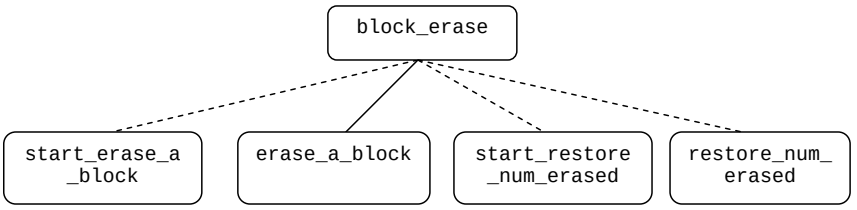


FIGURE 6.11: An event-refinement diagram representing an erasing process

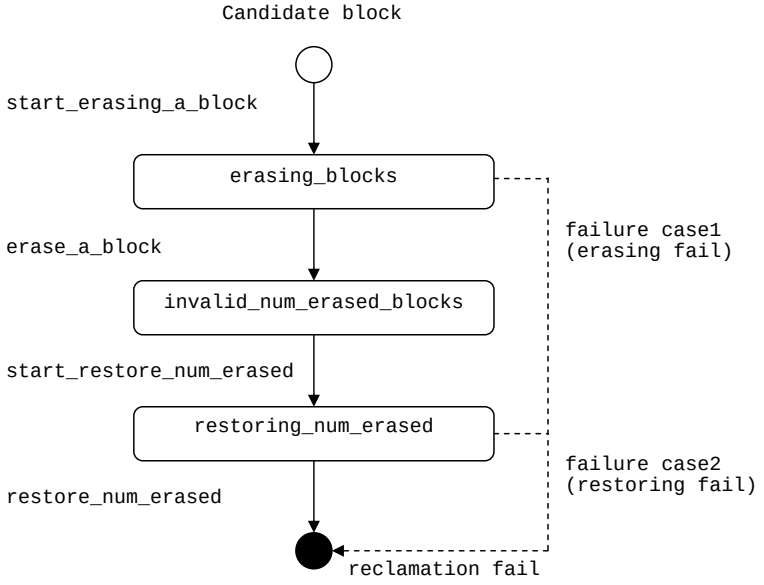


FIGURE 6.12: A state diagram representing states of blocks in erasing process

while others refine *skip*. Details of each event are given in Figure 6.13 and Figure 6.14. A state diagram of blocks in the reclamation process is shown in Figure 6.12. The selected block will be set to be in the erasing state when the *start_erasing_a_block* event is performed. The process of restoring the number of erasures will take place once the given block has been erased and is in the *invalid_num_erased* state. When the valid number of erasures has been restored, this block becomes a fresh block and is ready to be reused.

The reclamation events are divided into two phases given in Figure 6.13 and Figure 6.14. In order to start erasing any block (*start_erase_block*), we select an obsolete block with the least number of erasures (see *grd*1 and *grd*7) in order to balance the number of erasures across the blocks. (Similarly for the *start_relocate_a_block* event, we also select the candidate with the least number of erasures.) This method is a basic algorithm of the wear-levelling technique [62]. In phase1, the start event sets the state of the given block to be in the *erasing_blocks* state (*act*1), and writes the current number of erasures of the given block to a free page in another block that is not in the erasing process (*act*2 and *act*3). Secondly, *erase_a_block* erases the block (all pages are set to the default state) and sets the state of the given block to be *invalid_num_erased_blocks*. In phase2, the *start_restore_num_erased* sets the block with an invalid number of erasures to be

Event *start_erase_block* $\widehat{=}$
  Any $b, free\_r$ **Where**
    $grd1 : b \in obsolete\_blocks$
    $grd2 : b \notin erasing\_blocks \cup bad\_blocks$
    $grd3 : num\_erased(b) \leq max\_erase$
    $grd4 : free\_r \in RowAddr \setminus (programmed\_pages \cup programmed\_pages2)$
    $grd5 : BlkOfRow(free\_r) \notin erasing\_blocks$
    $grd6 : free\_r \notin dom(tmp\_num\_erased)$
    $grd7 : \forall k \cdot k \in obsolete\_blocks \setminus bad\_blocks$
        $\Rightarrow num\_eraseOfblock(k) \geq num\_eraseOfblock(b)$
  Then
    $act1 : erasing\_blocks := erasing\_blocks \cup \{b\}$
    $act2 : tmp\_num\_erased(free\_r) := num\_erased(b)$
    $act3 : corresBlkOftmpErased(free\_r) := b$
  End
Event *erase_a_block* **refines** *block_erase* $\widehat{=}$
  Any $rows, b$ **Where**
    $grd1 : rows \subseteq RowAddr$
    $grd2 : b \in erasing\_blocks$
    $grd3 : rows = BlkOfRow^{-1}[\{b\}]$
    $grd4 : rows \cap dom(trans\_func) = \varnothing$
  Then
    $act1 : flash := flash \lhd\!\!\!- (rows \times \{dp\})$
    $act2 : programmed\_pages := programmed\_pages \setminus rows$
    $act3 : obsolete\_pages := obsolete\_pages \setminus rows$
    $act4 : programmed\_pages2 := programmed\_pages2 \setminus rows$
    $act5 : obsolete\_pages2 := obsolete\_pages2 \setminus rows$
    $act6 : invalid\_num\_erased\_blocks := invalid\_num\_erased\_blocks \cup \{b\}$
    $act7 : obsolete\_blocks := obsolete\_blocks \setminus \{b\}$
    $act8 : erasing\_blocks := erasing\_blocks \setminus \{b\}$
  End

FIGURE 6.13: Reclamation process phase1: erasing a block

in the *restoring_num_erased* state. The *restore_num_erased* event restores the number of erasures to the (erased) block by increasing it by one, and then resets the state of the block being restored.

As already mentioned earlier, we also deal with faults in this development. Failures may occur at any points (as specified in Figure 6.15) – first is at the erasing process and second is at the restoring number of erasures. In the first case, the block is still in the obsolete state which is a candidate that may be selected to be erased later when reclamation is required. In the second case, the given block has completely been erased but the number of erasures has not been restored yet. In this case, this block still have an invalid number of erasures, since the *invalid_num_erased* flag has been set. However, the number of erasures of the block which is stored in another block still remain and

Event *start_restore_num_erased* $\widehat{=}$
  Any b **Where**
    $grd1 : b \in invalid\_num\_erased\_blocks$
    $grd2 : b \notin restoring\_num\_erased$
  Then
    $act1 : restoring\_num\_erased := restoring\_num\_erased \cup \{b\}$
  End
Event *restore_num_erased* $\widehat{=}$
  Any b, row **Where**
    $grd1 : b \in restoring\_num\_erased$
    $grd2 : row \in dom(tmp\_num\_erased)$
    $grd3 : b = corresBlkOftmpErased(row)$
  Then
    $act1 : num\_erased(b) := tmp\_num\_erased(row) + 1$
    $act2 : restoring\_num\_erased := restoring\_num\_erased \setminus \{b\}$
    $act3 : tmp\_num\_erased := \{row\} \lhd tmp\_num\_erased$
    $act4 : corresBlkOftmpErased := \{row\} \lhd corresBlkOftmpErased$
    $act5 : invalid\_num\_erased\_blocks := invalid\_num\_erased\_blocks \setminus \{b\}$
  End

FIGURE 6.14: Reclamation process phase2: restoring the number of erasures

can be restored later.

Event *erase_a_block_fail_case*1 $\widehat{=}$
  Any b **Where**
    $grd1 : b \in erasing\_blocks$
  Then
    $act1 : erasing\_blocks := erasing\_blocks \setminus \{b\}$
    $act2 : restoring\_num\_erased := restoring\_num\_erased \setminus \{b\}$
  End

Event *restore_num_erased_fail_case*2 $\widehat{=}$
  Any b **Where**
    $grd1 : b \in restoring\_num\_erased$
  Then
    $act1 : restoring\_num\_erased := restoring\_num\_erased \setminus \{b\}$
  End

FIGURE 6.15: Reclamation of a block fail

6.6 5^{th} Refinement: Status Register

The status register has an important role to determine whether the flash device is ready or not. The flash device is ready for performing any operation if all LUNs within the flash device are ready. If the flash is not ready, no operations are allowed to be performed. The status register is also used to indicate whether the previous operation that has been performed succeed or not.

In the ONFI specification [51, 52], each LUN contains a status register (SR). The status register is represented in the standard as an array of eight bits with different meanings:

$$SR[0] = FAIL, SR[1] = FAILC, SR[5] = ARDY, SR[6] = RDY, SR[7] = WP$$

Positions 2-4 are reserved. $FAIL$, RDY (ready) and WP (write protection) are frequently used. $FAILC$ and $ARDY$ are valid only for the program cache operations, optional operations depending on the flash device. More details about optional operations can be found in [52]. In our development, we concentrate on only mandatory operations (such as page-read, page-write, block-erase, etc.). Thus, $FAILC$ and $ARDY$ are ignored in this work.

In this refinement, a status register for each LUN is introduced. The write protection (WP) is represented as a bit within the status register. This WP bit is allowed to be set or reset by the flash commands. Page-program and block-erase operations are not allowed to be performed on any LUN that have been write-protected. In our research, we have compared two approaches of specifying status values of the status register. The first is specifying as a state function mapping from each LUN the a status value. The second is representing status values as state sets, following the work of Butler and Yadav [31].

In the first approach, using state function, a machine variable representing the status of each LUN will be formulated as

$$lSR \in LUAddr \rightarrow STATUS$$

where $STATUS$ is defined as an enumerated set of possible values of the status register in a context. That is, $STATUS = \{RDY, nRDY, FRDY\}$ where RDY represents the ready status (the RDY bit is true), $nRDY$ means not ready (the RDY bit is set to be false), and $FRDY$ represents the status of which $FAIL$ and RDY bits are true.

Table 6.1 shows three significant states of the status register that we specify in this refinement. As previously discussed, $FAILC$ and $ARDY$ are ignored in our development. Thus only the RDY, $FAIL$ and WP bits are addressed. In addition, in the standard, if

the *RDY* bit is 0, other bits (except the *WP* bit) are invalid. Because the validation of the *WP* bit is not dependent on others, it is better to specify the state of the *WP* bit separately. If we were to include this bit, the number of possible states would be increased (i.e. six states are required). This would make model more difficult to manage.

TABLE 6.1: A table representing states of the status register

states	RDY	FAIL	(WP)
lready	1	0	(0,1)
lreadyfail	1	1	(0,1)
lnotready	0	-	(0,1)

In the second approach, using state sets, each possible states of the status register is specified as a state-set variable. Below shows the state-set variables we introduced to represent the status of each LUN.

lready represents a set of LUNs with *RDY* bit is set to 1. This means this LUN is ready for execution of another command.

lreadyfail (ready and fail) means both *RDY* and *FAIL* bits are set to 1. This case indicates that the previous command performed on the selected LUN has failed and the LUN is now ready for another command.

lnotready represents a set of LUNs that are not ready. The *RDY* bit is cleared to 0. This means all other status bits are invalid and shall be ignored.

wprotected represents a set of LUNs which are write protected. (The *WP* bit is set to be 1.) This kind of LUNs is not allowed to be programmed or erased. This state can overlap with above three states.

In our experiment, we have found that the second approach, specifying using state sets, makes proof simpler. Namely, the second approach led us to gain a higher degree of automatic proof. Although more proof obligations are needed to be discharged for the second approach, all are automatically discharged. This approach was chosen for our development.

Figure 6.16 shows additional variables introduced in this refinement. State sets previously mentioned are defined and constrained by *inv*5.3 up to *inv*5.8. Variable *t_status* represents the current status of the target flash (indicating that the target flash is ready (*RDY*) or not ready (*nRDY*) for the next command). It is global for the whole flash device. Invariant *inv*5.9 says that if the target flash is ready means all LUNs' statuses are ready.

$inv5.1 : t\_status \in STATUS$
$inv5.2 : wprotected \subseteq LUAddr$
$inv5.3 : lready \subseteq LUAddr$
$inv5.4 : lreadyfail \subseteq LUAddr$
$inv5.5 : lnotready \subseteq LUAddr$
$inv5.6 : lready \cap lreadyfail = \varnothing$
$inv5.7 : lready \cap lnotready = \varnothing$
$inv5.8 : lreadyfail \cap lnotready = \varnothing$
$inv5.9 : t\_status = RDY \Rightarrow (\forall l \cdot l \in LUAddr \Rightarrow l \in (lready \cup lreadyfail)$
...

FIGURE 6.16: Invariants of the third refinement

In this refinement, some extensions were made to some previous abstract events. For example, in order to start any new operation on a LUN, the status register of that LUN must be ready. Once the operation is started the status is set to be not-ready until the end of the operation. Figure 6.17 shows some changes made to the *write*2*reg* event in this refinement. For example, in case of success, the LUN being performed is moved from the not-ready state (*notready*) to the ready state (*lready*). Additional events related to status registers were also introduced, for example, *set_writeprotect*, *reset_writeprotect* and *read_status*. Details can be found in Appendix C.

On the other hand, if we were to specify using state function, the *w_start* event will be replaced by the specification given in Figure 6.18 where *lSR* is specified as a state function mapping from each LUN to a status value within *STATUS*.

At this point, we can see that although using state function does not make proof simpler, it makes the specification more readable and easier to model compared with specifying as state sets. Setting the value of the status register of each LUN is completed in one action, compared with the previous case that requires three (simultaneous) actions. This approach seems to be suitable if the number of states is larger. Thus, these two approaches are appropriate for particular cases. Developers may choose state function if there is a huge number of state values to specify, otherwise using state sets would be suitable. In our development, we chose state sets because we have only three state values and want to make proof of the model simpler.

6.7 Proofs

Proof statistics given in Table 6.2 show that 352 proof obligations were generated and all were discharged automatically by the Rodin tool. MCH_FL represents an abstract machine of the flash interface layer while MCH_R1 up to MCH_R5 represent its refinements. Note that the proof statistics of the machine MCH_R5 are based on using state

Event *pprog_start* **extends** *pprog_start* $\widehat{=}$
  Where
    $grd3 : t\_status = RDY$
    $grd4 : lid \notin wprotected$
  Then
    $act3 : lnotready := lnotready \cup \{lid\}$
    $act4 : lready := lready \setminus \{lid\}$
    $act5 : lreadyfail := lreadyfail \setminus \{lid\}$
  End
    ...
Event *pprog_end_ok* **extends** *pprog_end* $\widehat{=}$
  Where
    $grd4 : lid \in lnotready$
  Then
    $act5 : lready := lready \cup \{lid\}$
    $act6 : lnotready := lnotready \setminus \{lid\}$
  End

Event *pprog_fail* **extends** *pprog_fail* $\widehat{=}$
  Where
    $grd2 : lid \in lnotready$
  Then
    $act3 : lreadyfail := lreadyfail \cup \{lid\}$
    $act4 : lnotready := lnotready \setminus \{lid\}$
  End

FIGURE 6.17: Part of the fifth refinement focusing on page program

Event *pprog_start* **extends** *pprog_start* $\widehat{=}$
  Where
    $grd3 : t\_status = RDY$
    $grd4 : lid \notin wprotected$
  Then
    $act3 : lSR(lid) := nRDY$
  End

FIGURE 6.18: A refinement of the *pprog_start* event, in the case of using state function

sets. In the case of using state functions, we got total 56 POs. 48 of them were automatically proved while the rest are discharged interactively. (They may require more time (or powerful prover) to discharge automatically.)

We have got completely automatic proof for several reasons. First, based on experience of what we have learnt from the modelling of file system such as selection of formulation, we have analyzed possible forms of specifying flash properties before selecting one of them to model. For example, as discussed in Section 4.10.1 where the abstract flash

TABLE 6.2: Proof statistics of the flash model

Machines/Contexts	Total POs	Automatic	Interactive
MCH_FL	9	9	0
MCH_R1	66	66	0
MCH_R2	55	55	0
MCH_R3	56	56	0
MCH_R4	70	70	0
MCH_R5	142	142	0
Overall	398	398 (100%)	0

specification is introduced, we avoided using curried function to make model easier to specify and prove. We decided to use state sets instead of state functions (as example given in Section 6.6). An other example, as discussed in Section 6.2, we used the projection function to specify the row addresses and page registers instead of using cartesian product.

Second, in case of failing to prove any PO, that PO was used as a guideline to improve the model. That is, such PO will be checked to see why it cannot be discharged. In some cases, an additional guard needs to be added to the corresponding event in order to make the PO discharged automatically. Sometimes, additional invariants were required to discharge some POs.

Another reason is that the flash interface model is not too complex in proof, compared with the tree-structured file system model of Chapter 4. Many invariants specified in each level of the flash memory model are straightforward and easier to prove than the invariants specifying the tree properties.

We can see a huge number of POs to be discharged for the fifth refinement (MCH_REF5) because we needed to prove that all state sets are disjoint. In addition, it seems to be more POs to be discharged if there are more states. At this point, we have completed another experiment to compare. That is, instead of introducing $inv5.6$ up to $inv5.8$ to say that those state sets are disjoint, we could replace them by the following invariant

$$inv5x : partition(LUAddr, lready, lnotready, lreadyfail)$$

We have found that using this invariant could reduce the number of POs from 142 to 100, and all are still discharged automatically. That means, specifying state sets in this way would be more appropriate. Note that *partition* is a new operation that was added to Rodin towards the later part of our research.

6.8 Conclusion and Assessment

In this chapter, we have presented further refinements focussing on the flash specification after we have decomposed our model in Chapter 4. We began with investigating the ONFI specification, analysing and deciding which formulation is suitable for modelling each flash property. Incremental refinement was also used as our strategy to develop this model. Some useful techniques that we have learnt from the previous chapter of modelling and proof of the file system layer – such as careful section of formulation, using proof obligation as a guideline, etc. – were also employed in this experiment.

In the first level, we have only two main interfaces provided to the file system layer: *page_read* and *page_program*. After that, other requirements and constraints were later addressed in refinement steps. Namely, we first introduced page registers and partitioned the atomic events *page_read* and *page_program* in the first refinement. Relocating a page and erasing a block, processes required for block reclamation and wear-levelling technique, were introduced in next refinement steps. We have found that careful selection of formulation mentioned in the previous chapter and incremental approach are also useful for this case study. As it can be seen in Figure 5.1, we can achieve 100% proof obligations discharged automatically.

We have given another approach of specifying a sequence of events to be performed. Individual events (or sub-steps) can be introduced in an abstract level and later be ordered in the following refinement. In the level where they are specified, each step is non-deterministically chosen to be performed. In the refinement, we introduced additional flag/state variables which were used to formulate event' guards and control the sequence of events to be performed. We only prove that these individual events refine *skip* in order to show that executing them conform the previous abstraction. As already mentioned, this technique works when all steps of the process refine *skip*. An example can be seen in Section 6.3 (where we introduced individual steps required for the block reclamation process) and Section 6.4 (where we added sequencing control to force those steps to be performed in an order).

We also have completed some experiments to compare different styles of modelling. For example, modelling of page registers in Section 6.2 where projection function versus cartesian product; and modelling of status registers in Section 6.6 where state set versus state function. First, as discussed in Section 6.2, specifying PR using projection functions makes model more readable and easier to specify than using cartesian product. Namely, accessing an individual element within the cartesian product is more complicated. Second, modelling states of status registers as state sets lets us gain a higher degree of automatic proof, compared with specifying as a state function. However, specifying as a state function is easier to specify and read. As discussed in Section 6.6, modifying the state value can be done in one step if we specify using state function, while several steps are required if we specify using state set. However, these two ap-

proaches are suitable for each particular case. Developers may choose state function if there is a large number of state values to be specified otherwise using state sets would be suitable.

Additionally, based on experience of using the Rodin tool, comparing the previous release 0.8.x and the later release 1.x.x, some useful features which are extended make modelling easier. For example, considering the fifth refinement, release 0.8.x has no partition operation in Event-B, we need to add a huge number of invariants to clarify that all intersections between state sets are the empty set. Similar to the event-extension feature that have already discussed in Section 3.7, this feature is also useful for developing our flash interface model which is outlined in this chapter.

Chapter 7

Comparison with Related Work on Verifying Flash File System

7.1 Introduction

A number of formalisations of file systems have been developed by other researchers. Most of them are focused on file contents, and *read* and *write* operations. There is some work that deal with the structure of file systems such as a specification of a visual file system in Z by Hughes [78] and the work of Hesselink and Lali [70]. The work of Hughes is focused on a tree structure and operations affecting the tree structure, but file content and a manipulation of file content were not specified. The work of Hesselink and Lali is focused on modelling of a hierarchical file system using PVS [104]. This work, [70], covers basic file operations including move and remove directories. Another related work by Morgan and Sufrin presented in [101] is a specification of a Unix filing 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 concentrated 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 [58, 61, 60] have developed a verified model of the POSIX filestore accompanied by a representation and proof using the Z/Eves proof system [109].

Since the filestore challenge was proposed by Joshi and Holzmann [85] in 2005, other researchers have addressed this challenge, such as [34], [59], [54], [87], [33] and [115]. For example, Butterfield and Woodcock [34] have developed an abstract Z-specification of the ONFI standard [51]. There was no refinement and proof mentioned in [34]. Butterfield, Freitas and Woodcock [33] have followed the work given in [34] by adding more details focusing on the structural aspects of the flash devices together with proof using Z/Eves. Ferreira et al. [54] have developed and verified a VDM specification of the Intel Flash

File System Core [67]. Alloy [81] and HOL [64] were used as tools for model checking and theorem proving in [54]. They stated that this work has not been completed yet they still have difficulties of translating VDM to HOL. The work contributed by Kang and Jackson [87, 88] is a formal specification and analysis of a flash-based file system in Alloy. This work was focused on basic operations of a filesystem and features covering wear-levelling and fault tolerance. Another work developed by Taverne and Pronk [115] is a formal development of a POSIX-like file store using a flash memory. Promela [74] is the formal language used in [115] while model checking using Spin [75] is a mechanism used for verification. However, the wear-levelling was not covered in this work.

This chapter first gives an overview of related work in Section 7.2. A comparison covering particular points is given in the following section. Our work is compared with three pieces of related work that apply various methods (i.e. Alloy, VDM and Z) to the file store problem.

7.2 Related Work

Three related bodies of work are chosen for comparison with our work. First is the work of Kang and Jackson [87, 88] in Alloy; second is the work carried out by Ferreira et al. [54] in VDM; and third is the work of Freitas et al. [58] in Z. As mentioned in Chapter 2, VDM and Z are state-based approaches like Event-B that make it easier to compare. In the case of Alloy (a declarative language which is designed for model checking [81]), the features which are covered in the Alloy work are similar to our work. Namely, this work specifies read and write operations of both the file system layer and the flash interface layer, and also covers the wear-levelling process.

7.2.1 Alloy

The specification and analysis of a flash file system are described in [87]. This work demonstrated one abstract level of the POSIX file system which is later refined to link with the flash interface layer. This work focused on *read* and *write* operations. Other basic operations such as *delete* and *move* were not mentioned. This work did not focus on the tree structure. The location of each file is represented by a sequence of directory names. Two issues which were covered in this work are *wear-levelling* and *fault-tolerance*. In the case of the wear-levelling process, they described three steps of the reclaim procedure. First, look for a dirty block which contains obsoleted data and has the lowest erase count. Second, relocate the valid pages (that may exist) in the selected block. That is, rewrite the valid pages to new page locations (which are available) and then re-map to the new locations. Third, erase the selected block. This block becomes available to be reused. Considering fault-tolerance, this work focused on power loss recovery.

The specification was based on the mechanism described in the specification of the Intel Flash File System.

The Alloy Analyzer [81] was used as a model checker to check the refinement properties (which relate the abstract file system with the concrete file system, taking account of flash architecture) for *read* and *write* operations. This kind of verification is fully automatic within a finite scope. They stated that the total size of the file system they verified was 24 data elements (with 6 flash pages). The refinement properties were checked in approximately 8 hours. A number of iterations were used to correct the model when non-trivial bugs were found during the model-checking process. Note that Alloy does not have refinement built in. They manually defined the relationship between the abstract state-variables with the concrete state-variables together with assertion. Details can be found in [87].

7.2.2 VDM

The work given in [54] was aimed at specifying and verifying the Intel Flash File System Core [67] focusing on the file system layer. The flash interface and the low level layers were not covered in this work. A naming structure was used to define file locations instead of the tree structure (using parent function). The location of each file was represented by a sequence of directory names.

This work was carried out by using VDM as a formal language for specification. HOL [65] and Alloy were used for theorem proving and model checking respectively. In order to verify the model, the VDMTools [41] was used to generate POs and translate the VDM model into an HOL format for proof. Alloy played an important role to generate counterexamples to proof obligations, when there was a PO that could not be discharged by the prover. However, manual translation was needed to convert the VDM model to Alloy. In this work, some POs could not be discharged using the prover and had no counterexample found. They needed to prove these POs by hand.

This work was just started and has no refinement. They demonstrated one level of specification and its verification. A small set of features was addressed. This work did not mention which basic functionalities of a file system they covered. In the paper, they focused on only the *delete* operation covering delete file and directory, and showed how POs of these operations can be discharged. Other features, such as specification of the flash interface layer, were considered as future work.

7.2.3 Z

The work given in [58] is a Z specification and verification of the POSIX file system covering basic operations of a file system such as *read*, *write*, *create*, *delete*, etc. This

work is based on the specification of the Unix filing system developed by Morgan and Sufrin [101]. In this specification, instead of using a tree structure, the location of each object was formulated as a sequence of directory names, which is the path of each file. This work concentrated on file contents and naming operations used for manipulating these rather than structure manipulation operations such as directory *copy* and *move*. However, this work did not cover the specification of the flash interface layer.

The Z formal language was used to specify the model. Z/Eves was used as a tool for verification (that is, theorem proof). Proof statistics given in this work shows 1337 proof steps in total. Those were classified into *trivial* steps (48%) relying on automation rules included in Z/Eves, *intermediate* steps (34.8%) requiring knowledge of how Z/Eves conducts the transformation, and *creative* steps (17.2%) requiring domain knowledge of theorem proof such as instantiation.

In addition, there is another model of a flash memory specified in Z which was developed by Butterfield et al [34]. This work focussed on the ONFI specification [51]. Three main operations, *page-read*, *page-write* and *block-erase* were addressed. However, this work did not cover the specification of the file system layer (that involves basic file operations such as open, read and write a file). They presented one level of specification and no proof is mentioned in this work. This work, [34], has been refined by adding more design details of the flash structure in [33]. The work of Huges [78] is also a Z specification of a visual file system. In this specification, transitive closure was chosen to specify the main property of a tree structure, e.g. reachability. However, the no-loop property was not mentioned in this specification. In addition, refinement and proof were not given in [78]. Finally, we used transitive closure indirectly in order to make our model easier to prove, as already discussed in Chapter 4.

7.3 Assessment and Comparison

Besides different tools and methods used, key points which are selected to compare with the related work are discussed below.

7.3.1 Point 1: Features

Table 7.1 shows a comparison between our work and other related work consisting of the work in Z, Alloy and VDM. The specification of the file system we developed was based on the architecture of the Intel Flash File System, like the work in VDM. Our specification of the file system covers not only *read* and *write* operations like [87] but also basic operations such as *create*, *move* and *delete*, and access permissions. In addition, our work also cover a specification of the flash interface layer focusing on *page-read*, *page-program* and *block-erase* operations which are interfaces provided to the file system layer.

TABLE 7.1: Feature Comparison

Features	Event-B	Z [58]	Alloy [87]	VDM [54]
file system architecture	Intel	POSIX	POSIX	Intel
flash interface specification	ONFi	no	ONFi	ONFi
structure	tree-based	path-based	path-based	path-based
create	yes	yes	yes	yes
delete	yes	yes	no	yes
move	yes	yes	no	no
copy	yes	no	no	no
read,write	yes	yes	yes	no
open, close	yes	yes	no	no
truncate	no	yes	no	no
mkdir, rmdir	yes	yes	no	yes
permissions	yes	no	no	no
fault-tolerance	yes	yes	yes	no
concurrency	yes	no	no	no
flash operations				
page read/program	yes	no	yes	no
block erase	yes	no	yes	no
wear-levelling	yes	no	yes	no
executable implementation	yes	no	no	no

Compared with others, first, the work in VDM covered only the file system layer focusing on some basic operations such as *delete* (others, such as *read* and *write* operations, and the flash interface layer have not been specified yet). Second, the work in Alloy focused on only *read* and *write* operations. This work also covered the wear-levelling process and the fault-tolerance which is similar to our work. Third, the work in Z is a specification of POSIX file system focusing on basic functionalities for files and directories such as *create*, *open*, *read* and *write* operations. In this work they concentrated on the naming operation instead of *copy* and *move* directory. However, this work did not cover the specification of the flash interface layer.

The structure of the file system layer we modelled is the tree structure which is different from others. We have found that representing the tree structure as a parent function makes it easier to copy and move subtrees, compared with the naming structure (or path-based). For example, in order to move any subtree, only the parent of the root of the given subtree is required to be changed. On the other hand, if we were to represent the file structure as a path-based structure, the path of all objects belonging to the subtree must be changed.

Fault-tolerance, concurrency and wear-levelling process are three main issues that are addressed in our our development. It can be seen that concurrency were not addressed by others, fault-tolerance and wear-levelling were covered is some work. Unlike all the other work, an implementation of the model is also covered in our work. The aim of

this part is to show that our model is implementable following from the systematic translation rules proposed in Chapter 8.

Compared with other work, a specification of a visual file system in Z developed by Hughes [78] is similar to our work, since they used the tree structure as a representative of the file system. However, the specification of this work did not cover the no-loop property and has no proof supported. In our work, we have already proved that our model preserves the tree properties (no-loop and reachability properties). The work of Hesselink [70] is another work aimed at dealing with the hierarchical file structure covering making and moving directories. This work represents the file structure using path-based while our work used a parent function. However, [70] did not cover the flash specification. The work of Taverne and Pronk [115] is a POSIX-like file store using a flash memory. The structure of [115] is a path-based structure which is different from our work. Basic operations including files and directories manipulation were covered in [115] but the wear-levelling process was not addressed.

7.3.2 Point 2: Refinement strategy

In our work, an incremental refinement strategy is the main methodology used for our formal development. We used refinement to introduce new features in an incremental way to develop our models. After that (when all required features were addressed), structural refinement was used to refine the model by adding more design details to relate the specification of the file system to the flash specification. For example, we used the atomicity decomposition technique [31] as a mechanism to decompose an atomic event named *write-file* into *start-write*, *page-write* and *end-write*, in order to satisfy the *page-program* operation provided by the flash interface layer.

In our development, an incremental approach was chosen to make the model simpler and easier to prove. Namely, in each step, a small set of features is introduced, the complexity of modelling is reduced. Specifying everything in one level of specification makes models more complex and difficult to prove. For example, if we were to introduce files and directories in the same level as specifying the tree properties, then the *create* event would be replaced by events *crtfile* and *mkdir* because files and directories are different. Therefore, instead of proving only that the *create* event preserves the tree-properties, we would need to prove that both events, *crtfile* and *mkdir*, preserve the tree properties. In our approach, since we have already proved that the *create* event preserves the tree properties (in the abstraction), we do not need to prove it again (in the refinement) in order to show that events *crtfile* and *mkdir* preserve the tree properties if they are refined events of the *create* event.

A distinguishing feature of our treatment of the flash file system 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. As stated in [47], "*an abstract program [or specification] is, in general, easier to prove correct than a concrete one, this simplifies the structuring of the verification process*". Additionally, as presented in [44], the use of multiple levels of refinement makes the abstraction gap relatively small at each stage. That means the gluing invariants required for refinement verification are also relatively simple. We believe that this relative can ease proof effort. This is testified by the proof statistics we have given in Section 4.14 and 6.7. In addition, as we have already discussed in Chapter 4 and 5, the use of multi-levels also makes an evolution process of the model easier to carry out. Namely, if we were to specify everything in one level, this would make model more complicated and difficult to modify. In the case of multi-level approach, each level has its own individual purpose based on the features/requirements that have been introduced in that level. Thus, modification can be made directly to the level where the changes affect, then such changes will be propagated down automatically.

We also used the machine decomposition technique [26] to decompose our model into sub-models that can later be refined separately. As stated in [44], this is another distinguishing feature of our work. 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 flash file system 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 an important contribution of the thesis.

Compared with others, most of the related work has only one level of specification such as [34], [54], [78] and [115]. Another work given in [87] presented an abstract file system together with one level of refinement. It can be seen that an incremental refinement strategy is not the way they used to develop models.

7.3.3 Point 3: Verification Techniques

Theorem proving is a mechanism used for verifying our models like the work in VDM [54] and Z [58] while the work in Alloy [87] used model checking as a technology to analyse the model. [54] used theorem proving at the first step of verification. If some POs could not be discharged then model checking was used to analyse and find counterexamples. However, manual translation was needed for translating VDM to Alloy. Some POs were discharged by hand when counterexamples were not found. The details of proof statistics were not given in this work.

In the case of the Z model [58], they need to define proof scripts by hand before proving using Z/Eves. Based on proof statistics of this work shown in Table 7.2[1], more than 50%

[1]Note that superscript a represents the number of POs that were discharged automatically while superscript i represents the interactive proofs.

TABLE 7.2: Proof Comparison

Criteria	Event-B	Z [58]	Alloy [87]	VDM [54]
mechanism	proof	proof	model checking	proof+ model checking
tool	Rodin	Z/Eves	Alloy Analyzer	HOL + Alloy
number of POs	1069 (1028^a + 41^i)	219 (proof scripts)	na	na
total proof steps	577	1337	na	na
trivial steps	449 (78%)	642 (48%)	na	na
intermediate steps	43 (7%)	465 (35%)	na	na
creative steps	85 (15%)	230 (17%)	na	na

of the proof steps (consisting of 17.2% *creative* and 38.4% *intermediate*) are non-trivial. This seems to be that proving this model was not easy. Namely, a large number of interactive proving and knowledge in theorem proving are required. In this work, there is no statistics that make it clear about the number of proof scripts and steps which were discharged automatically by the tool.

In order to make a reasonable comparison, we classified the complexity of our proof steps into three categories (i.e. *trivial*, *intermediate* and *creative*) like [58]. In our circumstance (in Event-B and Rodin), *creative* steps cover particular kinds of proof steps such as adding hypothesis, instantiation, case distinction and proving by contradiction. Intermediate steps are simple kinds of interactive steps such as applying implication, removing negation, rewriting set equality, etc. Finally, trivial steps in our circumstance involve interactive steps that require little thought by user – such as simplification, trivial rewrites, equality substitution, etc.

Considering Table 7.2, in our development, a total 1069 of POs were generated by the Rodin tool. 1028 POs (96%) were discharged automatically while other 41 POs were discharged interactively. In case of automatic discharge, all proof steps required for discharging each PO are trivial and performed automatically by the Rodin tool. Proving the other 41 POs involves 577 proof steps. 449 (78%) of them are trivial steps, 43 (7%) of them are classified as intermediate steps. The rest, 85 steps (15%), are creative steps.

In case of trivial steps, they were performed automatically by the Rodin tool. Other types of proof steps (intermediate and creative) required interactive proving.

In addition, introducing additional invariants and theorems which are used for discharging POs or proving some system properties is also considered as a kind of creative step in Rodin. (In our development, 16 theorems were introduced to help proof.) The additional proved theorems we introduced can be reused to discharge some similar POs. For example, as discussed in Section 4.14, a tree-join theorem which was introduced in a context can be reused to prove that events *create*, *copy* and *move* preserve the tree properties. That is, instead of reproving the same thing in different events, this

technique makes interactive proof easier and saves the time required for proving.

7.4 Summary

We have outlined an overview of related work together with detailed-comparison with three pieces of related work in Alloy, VDM and Z. The features which were covered in each work are partially different from each other (see Table 7.1). For example, our work was focused on the tree-structure while others were based on naming-structure (or path-based). Concurrency was addressed in our models while it was found in other related work. Our work covered both the file system layer and the flash interface layer. Theorem proving is our methodology used for verification like the work in Z and VDM. An incremental refinement was used as a main strategy in our formal development which is different from others. As already discussed in Section 7.3.2, we have found that this approach can make models easier to specify and manage (e.g. modification of models). Additionally, we found that multi-levels of refinement also help evolution of the models, as already discussed in Chapter 5.

Chapter 8

Systematic Translation of Event-B Models into Java Code

8.1 Introduction

This chapter is aimed at outlining rules for translating Event-B specifications into Java code. We follow an object oriented programming approach. We provide systematic translation rules focusing on class construction and event translation together with examples that we believe other developers can learn from. The examples are based on the flash file system that have already discussed in previous chapters.

We chose Java because we can preserve modelling structure in Java. Java supports an object oriented programming that we follow. Java is an object oriented programming language providing many features that are useful for system development, such as reusability, polymorphism, inheritance, etc [42]. The Rodin toolset and most of plug-ins are based on Java-Eclipse. We believe that our rules would be easier to collaborate with others in future.

The rules we propose here are aimed at general use, not just for the file system model. However, limitations still remain. Some lines of Event-B cannot be translated using our rules (these were translated individually by hand). At the moment, we could not define all possible rules for translating Event-B, but we have given some guidelines to be followed in general. Details are discussed in Section 8.26.

This chapter begins with rules for constructing classes from the Event-B specification (in Section 8.2) before focusing on event translation in Section 8.3. Related work is discussed in Section 8.4 and Section 8.5 concludes the chapter with some discussion.

8.2 Class Construction

To describe how classes are constructed, we divide our explanation into two categories based on where classes come from: (i) contexts and (ii) machines. These two categories are the main structures that can be used to construct classes. In the case of contexts, each defined type is considered to become a class. In the case of machines, we have two sub-categories. First is a set machine classes, which are constructed from the machine themselves. Second is a set of internal classes, which are constructed from machine variables.

8.2.1 Defined Types in a Context as Java Classes

A general rule for translating a defined type into a Java class is given in Figure 8.1. Suppose we have a defined type named ST specified in a context given on the left of Figure 8.1. Constants $p1OfST$, $p2OfST$, ..., $pnOfST$ represent the properties of ST, which are specified as total functions mapping to the type of each property. Each Tx can be a user-defined type or a general type such as $\mathbb{N}$, $BOOL$, etc. In the implementation, based on the ST specified, we will get a class representing this type on the right of Figure 8.1. The result class is called ST and includes an attribute for each property of ST defined in the context.

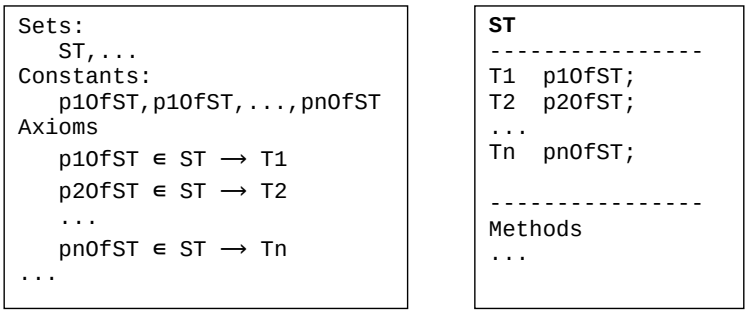


FIGURE 8.1: Rule 1: Converting a defined type to a class

Additional methods should be introduced in this class in order to get and set the value of each attribute, as an example given in Figure 8.2. However, they are not required for all attributes. For example the set method is not required for the static properties that are not allowed to be modified, such as a block id of each row address, etc.

Note that if there are no specific properties specified for any defined type, this type will become a class with no specific attributes.

To illustrate the application of Rule 1, we chose part of our flash file model to be applied. In our development, we have several data types, such as $OBJECT$, $USER$, $GROUP$, $PDATA$ and $RowAddr$, which are defined as carrier sets in contexts. These

```
public T1 get_p1OfST( ){
     return p1OfST;
}
```

FIGURE 8.2: A get-method of *ST*

types represent records which are specified as projection function. Figure 8.3 gives an example of *PDATA* and *RowAddr* type-specification. *PDATA* is composed of object id, page index, version number and data, while *RowAddr* is composed of LUN id, block id and page id. Figure 8.4 shows classes (*PDATA* and *RowAddr*) which are constructed

Sets
$PDATA, RowAddr, ...$
Constants
$objOfpage, pidxOfpage, versOfpage, dataOfpage$
$lidOfRow, bidOfRow, pidOfRow, ...$
Axioms
$objOfpage \in PDATA \rightarrow OBJECT$
$pidxOfpage \in PDATA \rightarrow \mathbb{N}$
$versOfpage \in PDATA \rightarrow \mathbb{N}$
$dataOfpage \in PDATA \rightarrow DATA$
...
$lidOfRow \in RowAddr \rightarrow \mathbb{N}$
$bidOfRow \in RowAddr \rightarrow \mathbb{N}$
$pidOfRow \in RowAddr \rightarrow \mathbb{N}$
...

FIGURE 8.3: A context representing part of defined types

from the definition specified in Figure 8.3 to which Rule 1 is applied.

```
PDATA
------------------
OBJECT objOfpage;
int pidxOfpage;
int versOfpage;
DATA dataOfpage;
...
```

```
RowAddr
-----------------
int lidOfRow;
int bidOfRow;
int pidOfRow;
...
```

FIGURE 8.4: Classes implementing PDATA and RowAddr

8.2.2 A Machine as a Class

Figure 8.5 gives a general rule for constructing a class from a machine named *MCH*. To translate a machine, the machine itself becomes a class containing attributes, which are

constructed from machine variables and their corresponding typing invariants. Machine

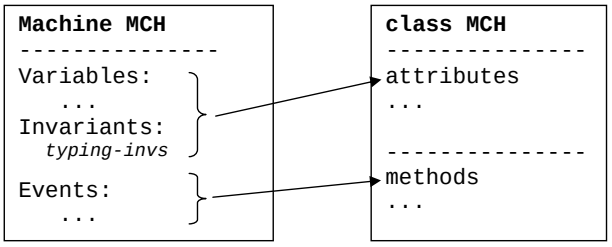


FIGURE 8.5: Rule 2: Translating a machine into a class

events are implemented as methods of the machine class. Details of the transformation of events into Java code will be addressed later in Section 8.3.

As given in Figure 8.5, to implement attributes of a machine class, only typing invariants and related variables are selected to define those attributes. We proposed two major sub-rules for translating two different types of machine variables: (i) a simple variable which is specified as an element of a set and (ii) a set variable which is specified as a collection of data. Details are explained below.

First, we have a variable named b which is specified as a single element of B, as can be seen from Figure 8.6. Thus, in the implementation, b becomes an attribute (which is typed B) of the machine class named *MCH*.

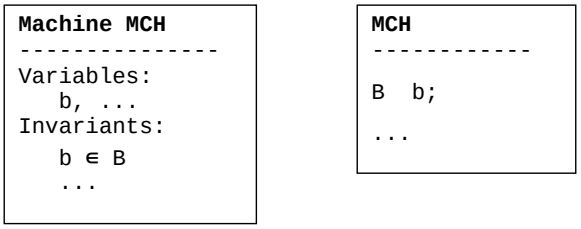


FIGURE 8.6: Rule 2a: Translating a simple variable in a machine class

Second, Figure 8.7 reveals that variable a is specified as a set of instances of A. This variable will become an attribute of the interface class which is implemented as a collection of instances of A. In Java, a collection of instances can be implemented by using arrays or other structures such as linked lists, trees, etc [56]. An array is simpler and easier to follow, compared with using linked list which is more complicated but flexible for memory allocation (and flexible for unbounded lists of instances). That means, each structure is suitable for a particular type of data collection. For example, if the number of instances is unbounded then the linked list is appropriate for implementing a list of these instances. On the other hand, if the maximum number of instances is known, implementing using array would be appropriate. In order to make our translation rules more general, we will not choose one of those, instead we will use the '*_Collection*' term to represent a collection of classes' instances. In the implementation, for example,

ACollection a; will be replaced by *A[] a;* if it is implemented using array, or *A a;* if it is implemented using a linked list where *A* is implemented as a linkable class[1].

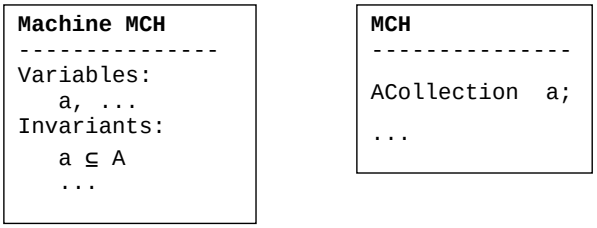


FIGURE 8.7: Rule 2b: Translating a set variable as a collection in a machine class

Figure 8.8 gives an extension rule for implementing a property which is specified as a partial function over natural numbers. In the machine, Variable *p* is specified as an array of *A* instances. In an implementation, *p* becomes an attribute of the machine class which is implemented as an array of *A* instances, as given on the right.

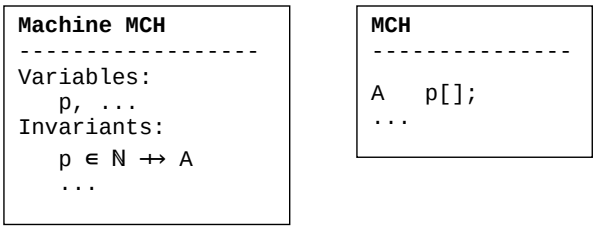


FIGURE 8.8: Rule 2x: Translating an array property

Note: In the case of machine decomposition, we may have several machine classes representing the entire system being developed. For example, if a machine is finally decomposed into *n* sub-machines then we will get *n* machine classes representing those *n* sub-machines. From Figure 8.9, the machine A is decomposed into A1 and A2. As given in the figure, A1 and A2 will become machine classes that interact with each other via shared events, as discussed in Section 4.13. Again if further machine decomposition is applied to the machine A1 to gain sub-machines A1a and A1b, then we will finally have three machine classes A1a, A1b and A2 representing such sub-systems being developed. (The A1 class is replaced by A1a and A1b.) In the implementation, each machine class interacts with its related machine by method calling. Details of implementing of shared events are given in Section 8.3.4.

8.2.3 Machine Variables as Classes

The purpose of this section is to explore systematic rules used for constructing internal classes from machine variables when additional properties are specified for them.

[1]A linkable class is a class that has at least one linking attribute representing the object which is next to itself. This attribute is typed as the class name. The number of linking attributes depends on the type of linked lists (i.e. one for a single linked list; and two for a double linked list) [56]

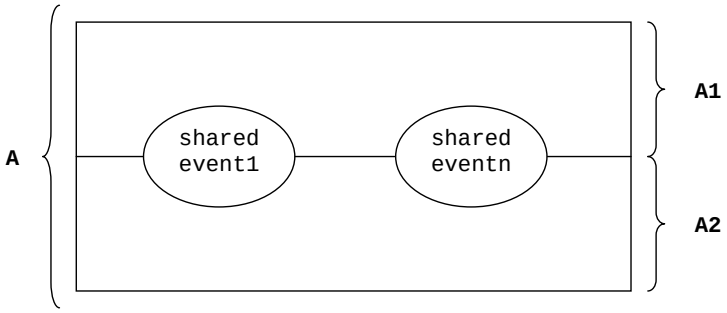


FIGURE 8.9: A diagram representing machine decomposition

First, from Figure 8.10, if *p1* specifies a specific property of variable a and is typed as *T1*, then a becomes a class named A and *p1* becomes an attribute of class A presented on the right hand side.

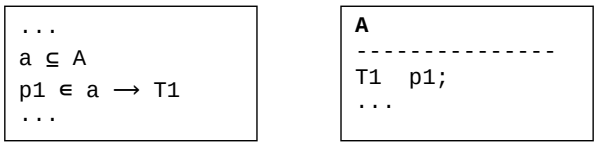


FIGURE 8.10: Rule 3: Function over a set variable

Second, from Figure 8.11, if c is a subset or equal to a and elements of sets can move in and out of c [and there are no specific properties for c], then c becomes a boolean state property of A. For example, *r_opened_files* $\subseteq$ *files*, a file element can be moved to the *r_opened_files* state when it is opened and moved back when it is closed. On the other hand, for example, *files* $\subseteq$ *objects*, when a file element is removed from *files*, it is also removed from *objects* totally. That means, in this case, *files* shall not be implemented as a boolean attribute. The appropriate way to implement *files* is to construct *files* as a subclass of the *objects* class which is discussed later in Rule 5.

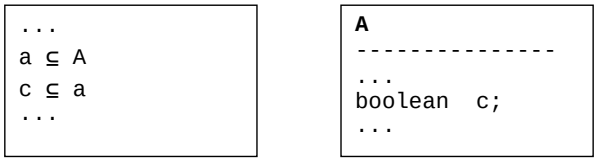


FIGURE 8.11: Rule 4: Subset of set variable as a boolean property

Third, from Figure 8.12, if we have *p1*, c and d specifying specific properties of a like the previous case, but we also have specific properties for c and d then we will get a result given on the right. Namely, c and d are constructed as sub-classes of A, named C and D respectively. *p1* is a common property of both C and D, while *p2* and *p3* are specific properties of C and D respectively.

Fourth (as can be seen in Figure 8.13), if we have an invariant saying that $p5 \in c \leftrightarrow T5$, this means an element of c may have more than one corresponding value of *T5*. This

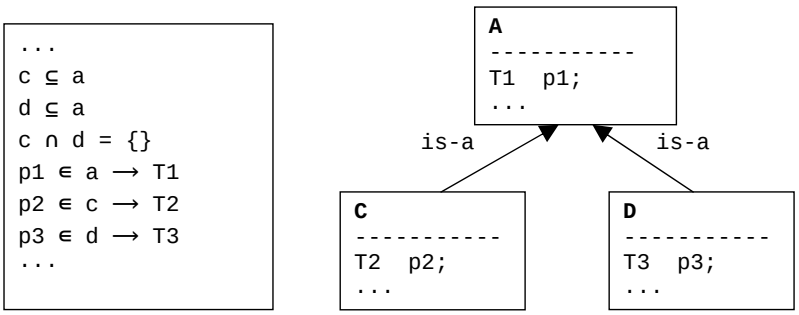


FIGURE 8.12: Rule 5: Subset to sub-classes

kind of property will be implemented as a collection of *T5*' instances as given on the right of the figure.

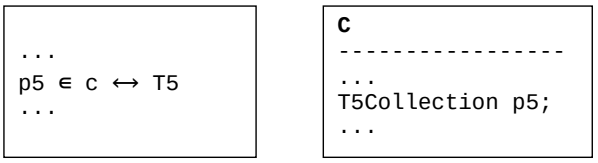


FIGURE 8.13: Rule 6: Relation to a list-attribute

Fifth, if any property is specified using a partial function (rather than a total function), we have added an additional rule to deal with this as given in Figure 8.14. This partial function means that not all elements of *c* have this property. Not only is *p6* implemented as an attribute of *C*, but we also have an additional flag attribute to indicate that *p6* is valid only when this flag true. For example, getting the value of *p6* succeeds only when *existP6* is true.

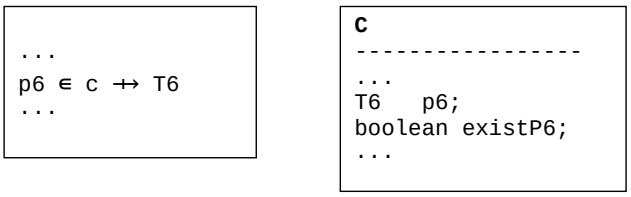


FIGURE 8.14: Rule 7: Partial function to class property

Figure 8.15 gives a rule for translating any property which is specified as a partial function over numbers. On the left, *p7* is a property of *a* which is specified as an array of elements typed *T7*. The right hand side shows the *A* class where *p7* is implemented as an array of *T7*. The domain of the array property should be contiguous although the specification ($\mathbb{N} \pfun T7$) is defined as a partial function. If the domain is not contiguous, implementing as a dynamic structure such as a linked list would be more appropriate.

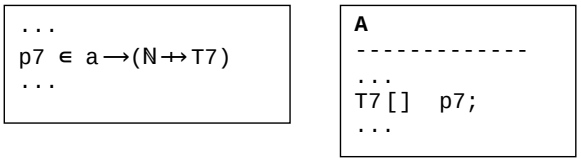


FIGURE 8.15: Rule 8: Translating an array property

8.2.4 Application of Rules

In our development, we have two main variables specified in the file system machine: *files* and *directories* (others such as *users*, *groups*, etc., will not be focused on in this report). These two variables are implemented as attributes of the machine class, named *FILEMCH*. The machine class we get is given in Figure 8.16 where Rule 2b is applied to set variables *files* and *directories*. Namely, *files* and *directories* are implemented as collections of files and directories, respectively.

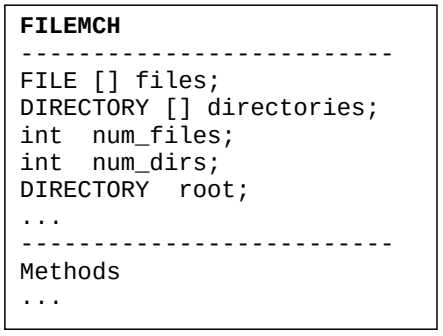


FIGURE 8.16: A machine class representing the file system model

In this model, we also have $root \in directories$ specified as a machine invariant. Thus, *root* is implemented as an instance of *DIRECTORY* in the *FILEMCH* class, where Rule 2a is applied. Attributes *num_files* and *num_dirs* are used to keep the number of files and directories, respectively. In Event-B, these two properties are specified as

$$num\_files = card(files)$$
$$num\_dirs = card(directories)$$

Moreover, we also have other properties specified in the machine. Some are specific to files, and some are common to both files and directories. Below are examples of applying translation rules that have been discussed previously.

In our development, we specified a set of objects as *OBJECT* instances. Each object can be either a file or a directory. Figure 8.17 shows some of machine invariants modelling these properties. From the specification given in Figure 8.17, we get a class diagram representing classes *OBJECT*, *FILE* and *DIRECTORY* given in Figure 8.18. As previously mentioned, when an element of files or directories is removed, it is also removed

$$
\begin{array}{l}
objects \subseteq OBJECT \\
files \subseteq objects \\
directories \subseteq objects \\
files \cup directories = objects \\
files \cap directories = \varnothing
\end{array}
$$

FIGURE 8.17: Part of machine invariants defining *objects*, *files* and *directories*

from the set of objects, that means these *files* and *directories* shall become subclasses of the *objects* class. In addition, we have specific properties for files and directories, thus Rule 5 is applicable to this case. (Note that we use UPPERCASE to represent classes.)

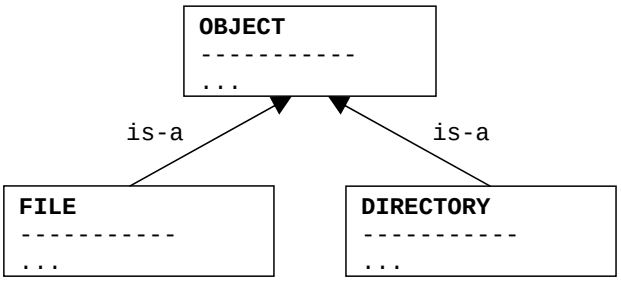


FIGURE 8.18: A class diagram of OBJECT, FILE and DIRECTORY

The *OBJECT* class is the generalized class of *FILE* and *DIRECTORY*. The common properties (or attributes) of each object are specified as a number of machine invariants shown in Figure 8.19. These properties lead us to obtain an *OBJECT* class with at-

$$
\begin{array}{l}
parent \in objects \setminus \{root\} \rightarrow directories \\
obj\_name \in objects \rightarrow NAME \\
createdDate \in objects \rightarrow DATE \\
lastModiDate \in objects \rightarrow DATE \\
obj\_owner \in objects \rightarrow users \\
obj\_group \in objects \rightarrow groups \\
obj\_perms \in objects \leftrightarrow PERMISSION
\end{array}
$$

FIGURE 8.19: Part of machine invariants defining objects' properties

tributes given in Figure 8.20 where Rule 3 and Rule 6 are applied. Note, because the *obj_perms* (objects' permissions) is specified as a relation – that means one object may have more than one permission types (based on its owner, group and world) – we implement it as an array of permission types. Additionally, as mentioned earlier, additional methods for setting and getting the value of each attribute are also required.

The specific properties of files which are specified as machine invariants are given in

```
OBJECT
--------------------
DIRECTORY parent;
NAME obj_name;
DATE createdDate;
DATE lastModiDate;
USER obj_owner;
GROUP obj_group;
PERMISSION [] obj_perm;
...
```

FIGURE 8.20: An OBJECT class

Figure 8.21.

$$fat \in files \rightarrow (N \nrightarrow RowAddr)$$
$$fsize \in files \rightarrow N$$
$$current\_version \in files \rightarrow N$$
$$...$$

FIGURE 8.21: Part of machine invariants defining files' properties

File status values are also specified as state sets given in Figure 8.22. Note that, in an alternative way, we could specify the status values as total functions given in Figure 8.23.

$$w\_opened\_files \subseteq files$$
$$writing\_files \subseteq w\_opened\_files$$
$$r\_opened\_files \subseteq files$$
$$reading\_files \subseteq r\_opened\_files$$

FIGURE 8.22: Part of machine invariants defining files' status, type1

$$w\_opened\_files \in files \rightarrow BOOL$$
$$writing\_files \in w\_opened\_files \rightarrow BOOL$$
$$r\_opened\_files \in files \rightarrow BOOL$$
$$reading\_files \in r\_opened\_files \rightarrow BOOL$$

FIGURE 8.23: Part of machine invariants defining files' status, type2

The reason we chose the first approach (using state sets) is to make proof simpler (as already discussed in Chapter 6). However, both led to the same result, as specified as properties of class FILE described in Figure 8.24, where Rule 3 and Rule 8 are applied.

```
FILE extends OBJECT
--------------------
RowAddr[]  fat;
int  fsize;
int  curr_version;
boolean  w_opened;
boolean  writing;
boolean  r_opened;
boolean  reading;
...
```

FIGURE 8.24: FILE Class

Figure 8.25 shows a DIRECTORY class where *members* represents a list of children belonging to each directory, while *dsize* represents the number of children of each directory.

```
DIRECTORY extends OBJECT
------------------------
OBJECT [] members;
int dsize;
...
```

FIGURE 8.25: DIRECTORY Class

Note that, so far, there are no specific properties for directories specified in the model. However, in Event-B, *members* and *dsize* could be specified as

$$members = parent^{-1}$$
$$dsize = directories \rightarrow \mathbb{N}$$
$$\forall\, d \cdot d \in directories \Rightarrow dsize(d) = card(members(d))$$

8.3 Event Transformation

In general, the reader may understand that all events specified in the machine will become methods of the machine class. The reader may have thought that one event must be translated to exactly one corresponding method. In fact, several events may be merged into one corresponding method, and an event may have more than one corresponding method. For instance, one event may have one method implementing the event itself and one separate method implementing one or more of its guards that return boolean values.

In this section, we divided event-transformation rules into several sub-sections covering basic events, event groups, event loops, shared events and concurrent events.

8.3.1 Basic Events

This section aimed at proposing general rules used for translating basic events to Java methods. Generally, in Event-B, an event is composed of four elements: name, parameter, guard and action.

$$evt\_name \mathrel{\widehat{=}} \textbf{any } i,x \textbf{ where } grd(v,i,x) \textbf{ then } act(v,i,x) \textbf{ end}$$

Parameters (i, x) are defined and constrained by the event guard (grd). Parameters specified here can be both internal (i) and external (x) parameters. However, in Event-B, there is no explicit distinction between internal and external parameters. Thus, we will impose this distinction through a naming convention (p_i for internal and p_x of external). The action act is performed only when the guard holds. Here v represents machine variables.

Simple rules of event transformation:

- An event name becomes a method name.
- A guard becomes a condition within a method and also the declaration of external and local parameters of the method.
- An action identifies the purpose of the method. It is expressed in terms of programming statements/instructions.
- In addition, some guards can be implemented as separate methods that return a boolean value. Similarly, an action of an event may be implemented as a separate method. These depend on styles of implementation and developer's preferences. If an action or a guard is complicated and difficult to express in one line of code, implementing as a separate method would be appropriate. Another reason, based on encapsulation concept[2] of the object oriented programming, instead of setting the value of any attribute directly, using method call would be more appropriate. Details and examples are given later in this section.

The scheme given in Figure 8.26 is aimed at introducing a prototype of event-to-code transformation. (Note that BF stands for "B Form" and JF stands for "Java Form".) Figure 8.26 (BF) represents an event structure that consists of parameters, guards and actions. Internal parameters become local variables of the Java method, while external parameters become method parameters. In this example, prm_i is specified as a local parameter and prm_x is specified as an external parameter. In the case of a local parameter, for instance, suppose we specified

[2] Attributes of classes should be defined as *private* or *protected* and their values are allowed to set or get via the methods provided [42]

$$grd1\_i : prm\_i \in typeOfprmi$$

then we will get the declaration of the local variable, *prm_i*, as

typeOfprmi prm_i; (implementing *grd1_exp*)

where *typeOfprmi* can be user-defined type, *int*, *boolean*, *String*, etc.

BF

```
Event evt_name  =
    Any prm_i, prm_x Where
        grd1_i &      //defines prm_i
        grd2_x &      //defines prm_x
        grd3 & grd4 &...& grdn //pre-conditions
    Then
        act1
        act2
        ...
        actn
    End
```

JF.1

```
public rtype mtd_name (type prm_x){

    grd1_exp //declaration of prm_i

    if(grd3_exp && ... && grdn_exp){
    //if the condition holds proceed
        act1_expression;
        act2_expression;
          ...
        actn_expression;
        return rvalue;
    }

    return err_value;
}
```

JF.2

```
public rtype mtd_name (type prm_x){

    grd1_exp  //declaration of prm_i

    if(!grd3_exp)  return err_value;
    if(!grd4_exp)  return err_value;
      ...
    if(!grdn_exp)  return err_value;
    //if all conditions hold proceed
    act1_expression;
    act2_expression;
      ...
    actn_expression;
    return rvalue;
}
```

FIGURE 8.26: General rules for event transformation

Additionally, we present two approaches to an implementation of an event guard (*grd3* up to *grdn*) in Java (see Figure 8.26 (JF.1) and (JF.2)). In the case of JF.1, we implement guards as a compound condition using '&&'. When these guards hold, meaning when the compound condition is true, the actions *act1* up to *actn* will be performed. Here *rvalue* represents the return value corresponding to the return type (*rtype*). Similarly, an error value (*err_value*) shall be returned if it fails. The return type may be implemented as *void* of which the return value is nothing. In another approach, JF.2, we implement in such a way that if any guard is not satisfied, the method will be forced to abort and return an error value indicating that executing this method failed. This approach would be useful if we want to return or report an error message for a particular case of such failures relating to each particular guard/condition. For instance, we may have a particular error message for *grd3* if this guard or condition is false.

Figure 8.27 gives additional rules for translating an event into Java code based on Event-B types. In the figure, we can see that '$\in$' and '$\subseteq$' can be used in both declaration and

condition. However, implementation of some types is non-explicit. For example, set and relation operations vary by the style of implementation. That means one line of B-code representing any action may be implemented in different ways. For example, the Event-B action $writing\_files := writing\_files \cup \{f\}$ means the given file f is set to be in the writing state. Thus, we can implement it by setting the writing flag of f to be true (by $f.setWritingFlag()$) or we can set the flag value to be true directly (without calling a method). Another example, if we have $wbuffer := \{f\} \lhd wbuffer$ meaning releasing the write buffer of f, then this action might be implemented as a separate method to release the write buffer and is called like $f.resetWBuffer()$. On the other hand, its write buffer might be released directly by setting it to be null as $f.wbffer = null$.

Notation	Example	Meaning	Java Implementation
Condition			
=	x = y	x is equal to y	x == y
≠	x ≠ y	x is not equal to y	x != y
>, <, ≥, ≤		other comparisons	>, <, >=, <=
∧, ∨		conjunctions (and, or)	&&, ||
Assignment			
:=	x := y	The value of x is assigned to be equal to the value of y	x = y;
Arithmetic operations			
+, -, *, /		plus, minus, multiply, divide	+, -, *, /

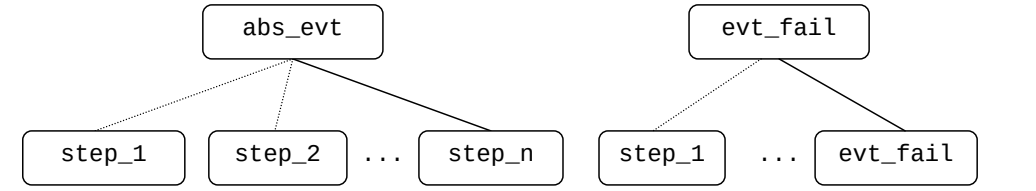
FIGURE 8.27: Additional rules for event transformation

Development of a comprehensive and systematic set of rules for set- and relation-operation translation is outside scope of this thesis. These depends on programming styles and developer's preferences. However, we proposed general rules for translating them with some examples.

8.3.2 Event Groups

The purpose of this section is to outline rules used for translating a group of events to a Java method. For instance, how can we implement decomposed events in Java? Figure 8.28 gives a general rule for translating a decomposed event into Java. The top of the figure shows an event-refinement diagram [26] representing an abstract event (*abs_evt*) which is decomposed into n sub-events: *step_1*, *step_2*, ..., *step_n*. The bottom represents two different approaches to the implementation of the decomposed event above. The first (on the left) is implemented as a simple form without separate methods. The second (on the right) is where sub-events are implemented as separate

methods. Based on the definition of the event-refinement diagram, sub-events will be performed in order from left to right. Thus, this corresponding method is implemented as a sequence of n steps (or sub-methods), which are ordered in the same way as the given specification. Namely, the *step*_1 event must be completed before performing *step*_2, and so on.




```
public rtype abs_mtd(ext_prms){
    declaration of int_prms;
    if ( JF(step_1.grd) ){
         JF(step_1.act);
    }else{
         JF(evt_fail.act);
         return err_value;
    }
    ...
    if ( JF(step_n.grd) ){
         JF(step_n.act);
    }else{
         JF(evt_fail.act);
         return err_value;
    }
    return rvalue;
}
```

```
public rtype abs_mtd(ext_prms){
    declaration of int_prms;
    if (JF(step_1)==false){
         JF(evt_fail);
         return err_value;
    }
    ...
    if(JF(step_n)==false){
         JF(evt_fail);
         return err_value;
    }
    return rvalue;
}
```

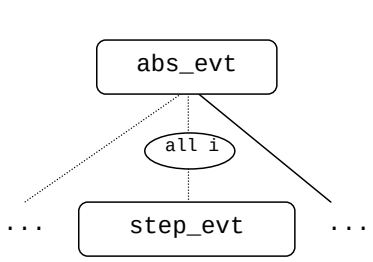
FIGURE 8.28: A general rule for implementing event groups

In the first approach presented on the left, the guard of each event is implemented as a condition of if-statement. If this condition is true, then the corresponding actions (in Java form) will be performed. If any condition is false, the Java form of the *fail_evt.act* will be performed and then return an error value. With the second approach showing on the right, each sub-event is implemented as a method returning a boolean value (or any value indicating whether the function being performed succeed or not). That is, if any method (representing each sub-event) failed, *JF*(*evt_fail*) will be performed and the error value will be returned to indicate that the process being executed failed. Finally, if all sub-methods have succeeded completely, the return value will be returned at the end.

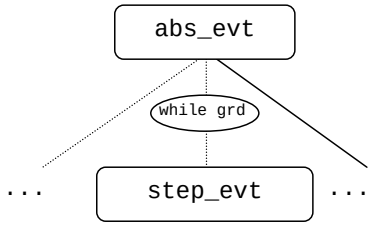
Note that we represent *JF* (Java Form) as a function that transforms the given B-specification into Java code. Suppose the return value of *JF(step_1)* is a method *public boolean step1_mtd()*{...}, we will get *if(step1_mtd()==false)* { ... *return err_value;* } in the first step. For instance, *JF*(*step*_1.*grd*) returns the Java form of the guard of the *step*_1 event.

8.3.3 Event Loops

There is no semantic notation for specifying loops in Event-B. However, loops can be modelled in Event-B. For instance, in the refinement diagram on the top-left of Figure 8.29, we use $all(i)$ to indicate that the *step_evt* event will be performed for all i before performing the event on the right. To specify a loop, additional flag/state variables are required and used to formulate guards to control the sequencing and iteration of the events to be performed. In this case, a guard – saying that the number of i that has been performed has not reached the number required – is needed to make the *step_evt* event enable. This condition will be false when all steps have been completed.




```
public rtype abs_mtd(ext_prms){
    ...
    for(i=init_val;i<=mx_val;i++){
        if(JF(step_evt.grd)){
            JF(step_evt.act);
        }
    }
    ...
}
```



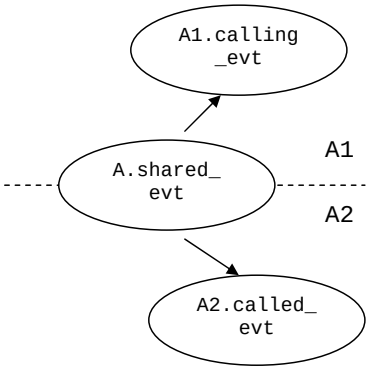

```
public rtype abs_mtd(ext_prms){
    ...
    while ( JF(step_evt.grd) ){
        JF(step_evt.act);
    }
    ...
}
```

FIGURE 8.29: A general rule for implementing event loops

In Figure 8.29, we present two styles of implementing event loops. First, a for-loop is used and then the second a while-loop is used. Each style is suitable for a particular case. Using a for-loop is applicable only when we have already known the number of iterations to be performed, otherwise using a while-loop is more suitable. On the top of Figure 8.29, we present an implementation of for-loop. In this case, the *step_evt* event will be performed for all i when the if-condition ($JF(step\_evt.grd)$) is true. Here *init_val* and *mx_val* represent a lower-bound and an upper-bound of the value of i. The bottom shows a loop event which is implemented using a while-loop. The guard of the *step_evt* event becomes a condition of the while-loop. All actions of the *step_evt* will be performed while the condition of the while-loop is true.

8.3.4 Shared Events

The purpose of this section is to provide a general rule for implementing a shared event synchronisation. As mentioned in Section 8.2.2, see Figure 8.30 where we have a machine named A which is decomposed into A1 and A2. Here *shared_evt* is a shared event of both A1 and A2 before decomposing. The shared event is split into two sub-events named *calling_evt* (of the machine A1) and *called_evt* (of the machine A2). Suppose the *called_evt* event of the machine A2 is called by the *calling_evt* event of the machine A1, we will get an implementation of the *calling_evt* as given on the right, which is located in the machine class A1. The *called_mtd* method representing the *call_evt* event is implemented on the A2 side (or in the machine class A2).




```
public rtype calling_mtd(ext_prms){
    [declaration of parameters]
    if !(JF(A1.calling_evt.pre_grd))
        return err_value;
    [rprm = ] JF(A2.called_evt(prms));
    if !(JF(A1.calling_evt.post_grd))
        return err_value;
    JF(A1.calling_evt.act);
    return rvalue;
}
```

FIGURE 8.30: A general rule for implementing a shared event

As given in Figure 8.30, when the *called* event is performed, a return value may be produced. We present *rprm* as a parameter to which the return value is returned. This parameter is a shared parameter specified for an interaction between these *calling* and *called* events. We classify the guard of the *calling* event into two parts. These two parts are *pre_grd* and *post_grd*, representing parts of the guard before and after calling respectively. Each of them might be omitted if they are not specified in the specification. For example, if the return value is not required for formulating any *post_grd* guard, the second part of if-condition will be omitted. When the *called* method has been completed and all conditions are satisfied, the action of the *calling* method will proceed.

It is noted that the composition in Event-B is symmetric [26]. There is no hierarchical or sequencing structure. That is, sub-components or sub-machines interact with each other via synchronisation over shared events. In Java, we implement the decomposed model as a hierarchical structure. For example, methods of A2 will be called by A1.

8.3.5 Concurrent Events

To implement events as concurrent methods, such events must be implemented as method classes implementing *Runnable* or extending *Thread* [63]. Figure 8.31 shows a scheme of implementing concurrent methods. Within the method constructor, an initial values of method variables will be assigned. The initial values depend on the values of passing parameters (*parms*). Method *run* is required for implementing this kind of method. The purpose of the given Event-B event is expressed within the body of method *run* in java form. The action of the event will be performed only when the guard of the given event holds. (Examples of and details of an implementation are given in Chapter 9.)

```
public class ccrt_mtd implements Runnable{
    [declaration of method variables];

    //method constructor
    public ccrt_mtd (parms){
        [assigning method variables];
    }
    //run method
    public void run(){
        if(!JF(ccrt_evt.grd)) return;
        JF(stepn.act);
    }
}
```

FIGURE 8.31: A scheme of implementing concurrent methods

In Java implementation, new thread will be created every time this kind of method is called. Several treads which are created will be run simultaneously. The Java code given below is an example of creating and starting a thread of the method name *ccrt_mtd*.

(new Thread(new ccrt_mtd(parms))).start();

8.3.6 Applying the Rules

In this section, we have three examples of event implementation. The first is a simple example of an implementation of the *incr_evt* which was given in Section 3.5. This event is composed of three sub-events: *start*, *step* and *end*. Figure 8.32 shows an implementation of this event where the rules given in Figure 8.28 and 8.29 are applied. This event has the effect of increasing the value of x by the value of y. Because there is no guard specified of the *start* event (see Figure 3.8), we have only an implementation of the action part of the *start* event.

Figure 8.33 shows an example focussing on the flash file model. This example shows an implementation of the *writefile* event which is also decomposed into three phases: *w_start*, *w_step* and *w_end*. Each sub-event is implemented as a separated method.

```
public boolean incr_mtd( ){

    int n;                          } Declaration of internal
    int x2;                           parameters
    boolean flag;

    n = 0;                          } JF(start)
    flag = false;
    x2 = x;

    while (flag == false && n < y) {
         x2 = x2 +1;                } JF(step)
         n = n + 1;
    }

    if(flag==false && n==y){
         x = x2;                    } JF(end)
         flag = true;
    }

    return true;
}
```

FIGURE 8.32: Java code implementing the *incr_evt* event

The third example is given in Figure 8.34. This figure shows the implementation of the *w_step* event where the rule of shared event synchronisation given in Figure 8.30 is applied. We implement the *w_step* event as a separate method because this process is complicated and will be called several times by the *writefile* method in order to write a file. Therefore, implementing as a separate method would be more appropriate. This method is located in the file system layer. In order to write a page, the *page_program* method provided by the flash interface will be called. If programming the given page succeeds, a true value will be returned. Otherwise a false value will be returned.

Note: If we want to implement events *writefile* and *w_step* as concurrent methods, these events must be implemented as method classes implementing *Runnable* or extending *Thread*, using the scheme given in Figure 8.31. Examples and details are given in Chapter 9.

8.4 Related Work

Recently, several code generators have been developed to translate formal specifications into programming code such as C++, Java, or Ada. For example, *B*0 of the B-method is a final refinement of a B specification that can be translated to programming code [90]. Atelier-B [38] is a tool that can generate C, C++ or Ada code from the *B*0. Java Card Code Generator [114], is another example of a code generator used to translate B specifications to Java code. The BART tool [107] is another tool for automatic refinement that is currently being developed by ClearSy. This tool aims at automatically refining the well-detailed B specification to a *B*0 implementation that can be translated to pro-

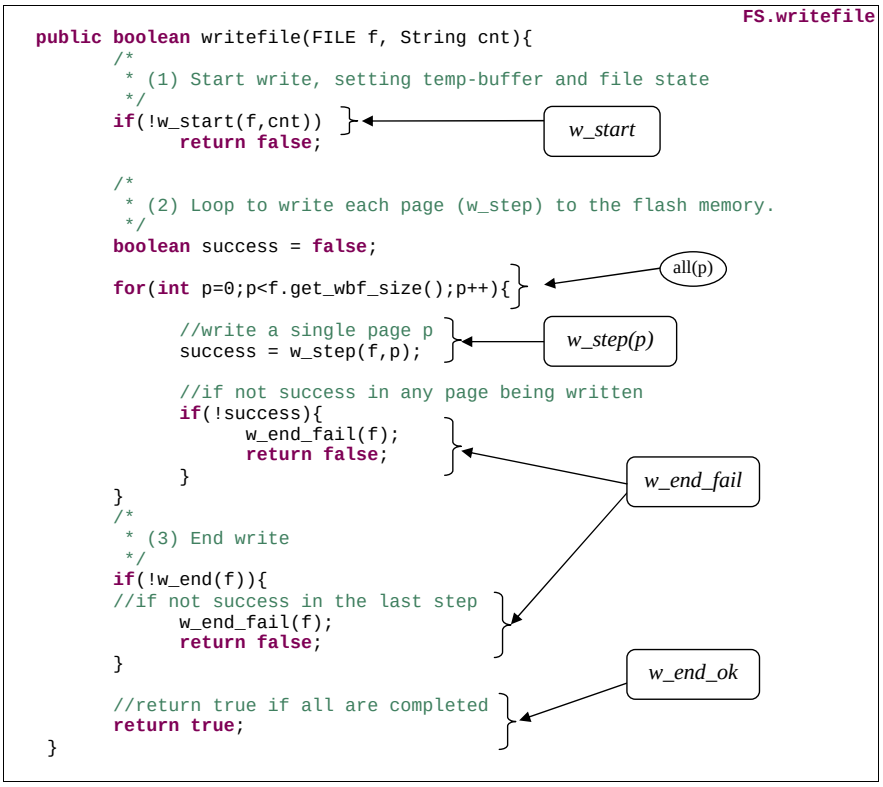


FIGURE 8.33: An abstract method *writefile*

gramming code using Atelier-B. Other extensions of existing formal methods such as Object-Z and VDM++ also have code generator tools. For instance, IFAD VDM [9] supports automatic generation of C++. However, most of code generated from those tools do not cover all system modules, such as the user interface which is manually implemented.

Considering Event-B, currently, there are no direct code generators published for Event-B. However, it has been considered as a future plug-in to be developed for the Rodin platform [7, 5]. Object-oriented Concurrent-B (OCB) [48, 49], is a recent project aimed at developing a tool used for designing models of concurrent systems. Each model is designed in the form of a Java-like model that can be automatically translated into an Event-B specification for verification and can be automatically translated into Java code for its implementation. Our work aimed at directly transforming an Event-B model into Java code which is different from the work of Edmunds (translating a Java-like model into Java code). The work of Wright [119] is to translate an Event-B model into C code, which is a structure programming style. This work focuses on event-transformation. In addition, UML-B [113] is one of the Rodin plug-ins allowing developers to construct UML-like models. This plug-in provides UML-like features, such as class diagram, state diagram, etc. These diagrams are automatically translated into Event-B specification. In contrast to [113], our approach is to construct Java classes from the Event-B specification, instead.

FS.w_step

```
      public boolean w_step(FILE f, int idx){
            //the given file must be in the writing state
            //f ∈ writing_files
            if(!f.isWriting()) {
                  return;
            }
            /*
             * Get the buffer content of the page specified
             */
            //idx|->cnt ∈ wbuffer(f)
            String pcnt = f.get_wbuffer(idx);
            /*
             * Construct a page data
             * pdata ∈ PDATA
             * objOfpage(pdata) = f
             * pidxOfpage(pdata) = idx
             * verOfpage(pdata) = writing_version(f)
             * dataOfpage(pdata) = cnt
             */
            PDATA pdata = new
                    DATA(file.getOID(),index+1,file.getWritingVersion(),pcnt);
            //r ∈ RowAddr
            /*
             * Call flash API to program this page where r is the return value
             */
            RowAddr r = f.page_program(pdata);

            if(r==null){ //if not succeeded
                  [Massage: Writing the given page fails];
            }else{       //if succeeded
            /*
             * Add new entry to the table of contents
             */
            // fat_tmp(f) := fat_tmp(f) ∪ {i ↦ r}
                  TOCEntry tocEntry = new TOCEntry(idx+1,r);
                  f.addToC_tmp(idx+1,tocEntry);
            }
      }
}
```

FIGURE 8.34: A method implementing the *w_step*

For a simple case (e.g. simple data structure), we might use OCB or Wright's code generators. However, for a complex structure (such as event decomposition), these tools may not appropriate. We may use a tool to translate an Event-B model into classical B specification in order to be converted into $B0$ that can later be translated to programming code. However, this requires more work. Additionally, code generators supporting $B0$-translation do not support concurrency [38].

8.5 Conclusion and Discussion

Systematic rules that can be used in general are important. These rules act as a framework or guideline for developers to implement the system model. In this chapter, we have proposed systematic rules for class construction and event translation that would be useful for developers in translating Event-B models into Java code.

While translating the Event-B specification to programming code, the readers may find that sometimes it is difficult to express lines of the specification into programming code because there are no specific rules for translation. As discussed in Section 8.3, one line of specification may be translated into different styles of programming code. However, those different styles must be a correct implementation. That means if we know what the

specification is or what the purpose of each line of the specification is, we will know how it can be coded using programming language even there are many styles to implement.

To make these translation rules more useful and applicable in the future, mechanical application of rules is important. For example, providing an automatic tool to refine the well-detailed specifications into the normal forms that are able to be applied by the translation rules.

For example, if we have

$$a \in \mathbb{P}(A)$$

specified in the specification, this line should be reformulated as a normal form as

$$a \subseteq A$$

in order to be able to be translated using Rule 2b.

Another example, if we have any guard specified as

$$grd : x \in A \setminus a$$

this guard should be split into

$$\begin{array}{ll} grd\_a : x \in A & \text{(declaration)} \\ grd\_b : x \notin a & \text{(condition)} \end{array}$$

which are the normal forms that are able to be translated using existing translation rules.

For a complex structure, such as

$$p \in x \rightarrow y$$

where x or y are a product of two entities or more.

For example, if $x = a \times b$, namely, $p \in a \times b \rightarrow y$, x could be reformulated as an Event-B record, where a and b are specified as its properties. Suppose this record type is named X in a context, we will get

$$p \in x \rightarrow y$$

where

$$x \subseteq X.$$

That is, Rule 3 is now applicable. Namely, X becomes a class with an attribute p typed Y, which is shown in Figure 8.35.

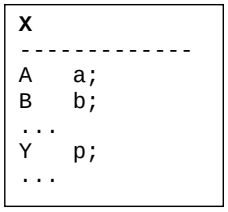


FIGURE 8.35: An example of class representing compound entity

Similarly, if y is also a product of some entities, such as $y = c \times d$, then we will get a class representing y where c and d are its attributes.

More systematic rules for implementing patterns of set and relation operations are also required for future work. As mentioned earlier, these kinds of operations (such as domain/range restriction, subtraction, etc.) may be implemented in various ways (depending on programming styles). Our work does not cover all Event-B notations but we provide the translation rules in general that developers can follow.

In addition, to guarantee the correctness of the translation rules, formal verification of translation (of both high-level specifications to low-level specifications (that are implementable) and low-level specifications to generated code) and verified translators are also important. We need proofs to ensure that the application of rules has been done correctly. Verification of generated code can be achieved by many approaches, such as providing an automatic code generator, or by providing assertions to verify the code, against. For example, in the case of JML (Java Modelling Language) [21], we may generate assertions from invariants and events specified in the models and then verify the code using the JML tool. These assertions will help developers to guarantee that Java code which are implemented preserves those required properties/assertions. An automatic translation tool is also required, to ensure that the assertions are formally generated and verified. Additionally, providing verified translators is an alternative approach to guarantee the correctness of code translation like Atelier-B translators [38] that aim to translate B0 (an implementable form of classical B) into programming code (e.g. C and Ada).

Chapter 9

An Implementation

9.1 Introduction

This chapter aims to present an implementation of a flash file system. The purpose of this implementation is to show how the model of flash file system can be implemented, how the translation rules can be applied, and to convince ourselves and the readers that the model we have developed is possible to be implemented.

Although we do not have an automatic code generator to translate the formal specification into programming code, it does not imply that the final implementation does not satisfy the given specification or is a bad implementation. To reduce the gap between the specification and its implementation, we have a set of systematic translation rules that have already discussed in Chapter 8. Our implementation of the flash-based file system presented in this chapter follows from the translation rules mentioned. (We manually transform our models into Java code following those rules.) However, as mentioned previously in Chapter 8, the rules we proposed do not provide all possible rules to cover all Event-B notation. For instance, lines of the specification related to set or relation operations (such as restriction, subtraction, overriding, etc.) vary by programming styles. Although we present no rules specific to these, we have provided general rules that may be followed.

As mentioned in Section 8.5, we may use code verification techniques – such as JML [21], automatic code generation – to guarantee the correctness of code. For example, we may generate assertions (in JML) from Event-B specifications and use JML tool to verify code. In addition, providing verified translators is another approach to ensure the correctness of code translation. This is another way of reducing the gap and gaining higher assurance. However, because of the time constraint, we did not use this technique for our implementation.

In this chapter, we begin with outlining a prototype of a flash file system in Section 9.2.

After that, a conclusion and future work are given in Section 9.3.

9.2 Prototype

Figure 9.1 shows an overview of the flash file structure which is summarised from the architecture of the Intel Flash File System Core Reference Guide [67]. In Figure 9.1, the file system layer is the part we specified in Chapter 5. We implemented this part as user/application interfaces providing commands to animate the file system. The flash interface layer down to hardware layer, the dashed box, is the part which is simulated. Some intermediate layers in the dashed box, such as the data object and the basic allocation layers are not mentioned in this figure. We assume that all layers within the dashed box are composed as one layer providing *page_read* and *page_program* interfaces to the file system layer. This simulation part corresponds to the flash model we specified in Chapter 6.

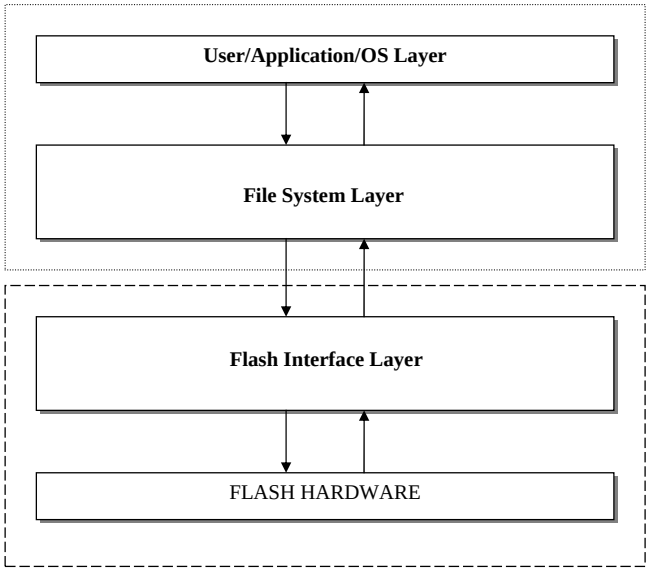


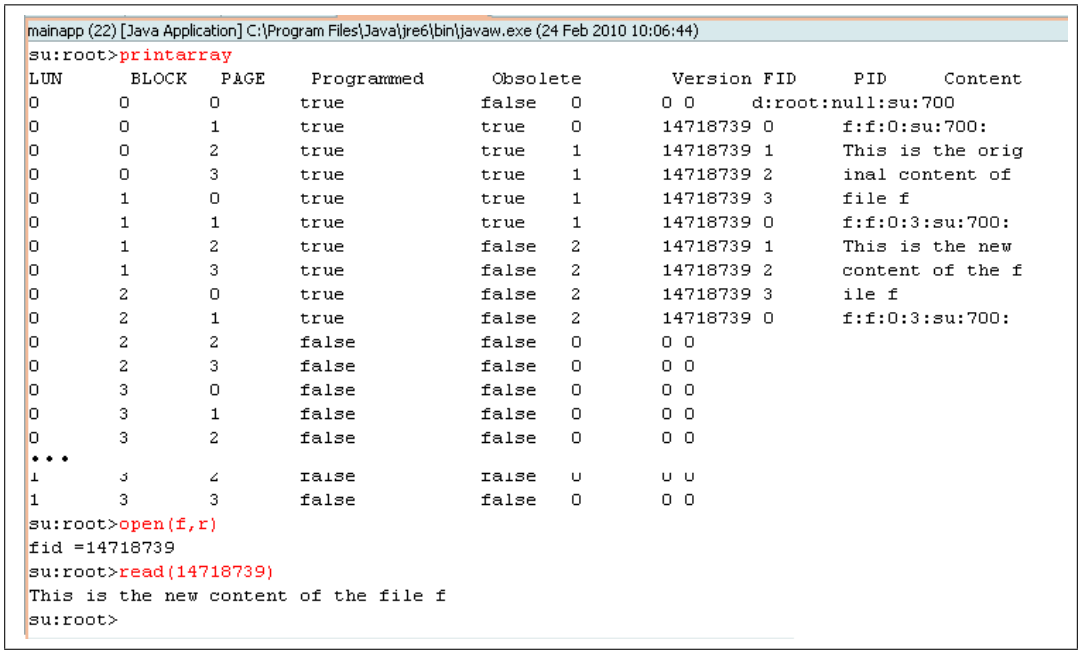
FIGURE 9.1: A structure of a flash file system

A prototype which is outlined in this section was implemented using Java on the Eclipse platform. Part of the simulation aims at simulating an array of pages within the flash memory and animating what changes are made to the flash array whenever file operations have been performed.

The reasons we chose to simulate a flash device rather than use the existing real flash are (i) simulation of faults is easier to be made since our work contrate on fault-tolerance, (ii) not all flash devices follow the ONFI standard we have followed, and (iii) because of the competition, underline specification of the recent products in the market are secret.

Figure 9.2 shows an example of flash simulation. It displays a current array of pages

within a flash memory. Each page is identified by a row address which is composed of a LUN address, a block address and a page address. Besides the address, each page contains its content and status fields which are used to identify the current status of each page, etc. In Figure 9.2, there are two versions of file f. The second version is valid while the previous version is obsolete. When the file is opened for reading, all pages corresponding to the valid version are read.



```
mainapp (22) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (24 Feb 2010 10:06:44)
su:root>printarray
LUN      BLOCK   PAGE    Programmed      Obsolete        Version FID     PID     Content
0        0       0       true            false   0       0 0     d:root:null:su:700
0        0       1       true            true    0       14718739 0      f:f:0:su:700:
0        0       2       true            true    1       14718739 1      This is the orig
0        0       3       true            true    1       14718739 2      inal content of
0        1       0       true            true    1       14718739 3      file f
0        1       1       true            true    1       14718739 0      f:f:0:3:su:700:
0        1       2       true            false   2       14718739 1      This is the new
0        1       3       true            false   2       14718739 2      content of the f
0        2       0       true            false   2       14718739 3      ile f
0        2       1       true            false   2       14718739 0      f:f:0:3:su:700:
0        2       2       false           false   0       0 0
0        2       3       false           false   0       0 0
0        3       0       false           false   0       0 0
0        3       1       false           false   0       0 0
0        3       2       false           false   0       0 0
...
1        3       2       false           false   0       0 0
1        3       3       false           false   0       0 0
su:root>open(f,r)
fid =14718739
su:root>read(14718739)
This is the new content of the file f
su:root>
```

FIGURE 9.2: A simulation of the flash array screen 1

Suppose in a reclamation process, block (0,1) has been reclaimed. Figure 9.3 shows a result of this. All valid pages at rows (0,1,2) and (0,1,3) have been relocated to rows (0,2,2) and (0,2,3) respectively. The old locations become obsolete and are ready to be erased. When this block (0,1) is selected to be erased, *programmed* and *obsolete* bits will be set to be false.

In the file system layer, file read and write methods are implemented as concurrent methods implementing *Runnable*. Figure 9.4 and Figure 9.5 show such implementation of the *writefile* and *w_step* events of the file system layer, where they are implemented in a concurrent style. Each call of *writefile* is a separate thread. Within a *writefile* thread, all *w_step* (write a page) are run concurrently.

Similarly, when the interfaces page-read and page-program (provided by the flash interface layer) are called, they can be executed concurrently in an interleaved fashion.

Methods accessing state variables within the flash, such as page-read and page-program must be synchronized in order to ensure thread safety [63]. Figure 9.6 gives an implementation of the *page_program* event which is an interface provided to the file system layer. Because this method has the effect of modifying the flash content that might

```
su:root>printarray
LUN      BLOCK   PAGE    Programmed      Obsolete        Version FID     PID     Content
0        0       0       true            false   0       0 0     d:root:null:su:700
0        0       1       true            true    0       14718739 0      f:f:0:su:700:
0        0       2       true            true    1       14718739 1      This is the orig
0        0       3       true            true    1       14718739 2      inal content of
0        1       0       true            true    0       0 0
0        1       1       true            true    0       0 0
0        1       2       true            true    0       0 0
0        1       3       true            true    0       0 0
0        2       0       true            false   2       14718739 3      ile f
0        2       1       true            false   2       14718739 0      f:f:0:3:su:700:
0        2       2       true            false   2       14718739 1      This is the new
0        2       3       true            false   2       14718739 2      content of the f
0        3       0       false           false   0       0 0
0        3       1       false           false   0       0 0
0        3       2       false           false   0       0 0
0        3       3       false           false   0       0 0
1        0       0       false           false   0       0 0
1        0       1       false           false   0       0 0

...
su:root>open(f,r)
fid =14718739
su:root>read(14718739)
This is the new content of the file f
su:root>
```

FIGURE 9.3: A simulation of the flash array screen 2

be accessed by several read/program operations simultaneously, this method must be *synchronized*.

9.3 Conclusion and Assessment

We have presented an implementation of a flash-based file system by using the translation rules given in Chapter 8. Our implementation covers two parts: (i) the file system layer providing interfaces to users and application programs and (ii) the flash interface layer providing interfaces to the file system layer. The first part implemented follows the specification given in Chapter 4. The implementation of this layer provides interfaces used to animate the system covering basic file operations such create, read/write files. The implementation of the second part follows the specification given in Chapter 6. This part aimed at simulating the flash device whenever the flash operations are performed.

The implementation we have completed still has a gap between the specification and the implementation. In order to narrow the gap, further work is still required – such as an automatic tool for refining Event-B models into the normal forms that are able to be translated using the translation rules, and automatic code generators.

Based on this implementation, much time was spent for specifying and reasoning about this model, but only few weeks were spent for coding the prototype. It can be believed that formal activities (modelling and verifying) make developers understand more about the systems being developed. As a result, the more understanding the developers have would make it easier to achieve an implementation of the system which satisfies the given specification. This could reduce the time used for the implementation as well.

FS.writefile

```
public class writefile implements Runnable{
    private int fid;
    private String cont;
    //constructor
    public writefile(int id, String cnt){
          fid = id;
          cont = cnt;
    }
    //run method
    public void run(){
          FILE f = getFile(fid);
          if(f==null){
                [Masseging: "File does not exist!"];
                return;
          }
          if(!f.iswOpened()){
                [Masseging: File has not been opened for writing!"];
                return;
          }
          /*
           * (1) Start write
           */
          w_start(f,cont);

          /*
           * (2) Write a single (w_step).
           * Loop to write each page to the flash memory.
           */
          for(int i=0;i<f.get_wbf_size();i++){
                try{
                      (new w_step(f,i)).run();
                }catch (Exception e){
                      writefile_fail(f);
                      return;
                }
          }
          /*
           * (3) End write
           */
          if(!w_end(f))
                writefile_fail(f);
    }
}
```

FIGURE 9.4: A concurrent implementation of the *writefile* event

However, it cannot ensure that the final implementation satisfies all what are specified in the given model. Therefore, additional systematic translation rules, verification of translation and automatic code generators are still required in the future.

FS.w_step

```
public class w_step implements Runnable{
     private FILE file;
     private int index;
     public w_step(FILE f, int idx){
           file = f;
           index = idx;
     }
     public void run(){
           //the given file must be in the writing state
           if(!file.isWriting()) {
                 return;
           }
          /*
           * Get the buffer content of the page specified
           */
           String pcnt = file.get_wbuffer(index);
           /*
            * Construct a page data
            */
           PDATA pdata = new
                   DATA(file.getOID(),index+1,file.getWritingVersion(),pcnt);
           /*
            * Call flash API to program this page
            */
           RowAddr r = flash.page_program(pdata);

           if(r==null){ //if not succeeded
                 [Massage: Writing the given page fails];
           }else{       //if succeeded
           /*
            * Add new entry to the table of contents
            */
                 TOCEntry tocEntry = new TOCEntry(index+1,r);
                 file.addToC_tmp(index+1,tocEntry);
           }
     }
}
```

FIGURE 9.5: A concurrent implementation of the *w_step* event

FL.page_program

```
   public synchronized RowAddr page_program(PDATA pdata){
         //r and pdata are shared parameters
         //r : RowAddr\programmed_pages
         //pdata : PDATA
         RowAddr r = getFreePage();
         if(r==null)
             return null;
         //extracting the row address
         int l=r.get_lid();
         int b=r.get_bid();
         int p=r.get_pid();

         //if the event guard satisfies
         //flash(r) := pdata
         //programmed_pages := programmed_pages ⩤ {r}
         flashArray[l][b][p]=pdata;
         programmed[l][b][p]=true;
         //return the address to which the given data is programmed
         return r;
   }
```

FIGURE 9.6: An implementation of the *page_program* event

Chapter 10

Modelling, Refinement and Proof Guidelines

10.1 Introduction

The purpose of this chapter is to contribute our modelling, refinement and proof guidelines. These guidelines are based on our experiments which were carried out using Event-B and Rodin to model a flash based file system. We firstly give modelling guidelines in Section 10.2. Refinement guidelines are given in Section 10.3 and proof guidelines are discussed in Section 10.4. Note that some improvement guidelines to the formal language and the Rodin tool are given in Chapter 11 where we assess Event-B and Rodin.

10.2 Modelling Guidelines

(MG1) Careful selection of formulation can ease proof effort. Based on experience of modelling the file system layer, in order to specify any system's property, possible formulations should be explored and analyzed to find which one is suitable for modelling that required property. For example, as explained in Chapter 4, we avoided using transitive closure to specify the no-loop property in order to find the easier way to prove. Instead of using transitive closure to specify no-loop property, we employed the no-loop theorem proposed by Abrial in [4] to model this property. As a result, this makes proof easier.

(MG2) Avoid using quantification to formulate specification, if possible. This could make models simpler and easier to proof. Namely, proving the preservation of invariants, which are specified using qualification, might require interactive instantiation that makes proof more difficult. To avoid, for example, instead of specifying an invariant like

$$inv\_a : \forall\, r \cdot r \in dom(trans\_func) \Rightarrow trans\_func(r) \in programmed\_pages2$$

it can be simplified as

$$inv\_b : programmed\_pages2 = trans\_func[programmed\_pages]$$

which is simpler, where $dom(trans\_func) \subseteq programmed\_pages$.

(MG3) Instead of introducing new machine invariants to satisfy some system properties, providing machine theorems and proving that these properties are satisfied is another mechanism used for specifying system models – for instance, modelling of the reachability property that we have already discussed in Chapter 4. We only need to show that machine theorems follow from the existing invariants and axioms. However, this approach can only be used when the new theorem follows from existing invariants.

(MG4) As partially discussed in Chapter 4, providing additional parameters in each event is useful sometimes. Although more guards are needed, it could make models more readable and easier to manage in both specifying and proof. Moreover, it is felt that providing a separation of input and output parameters (e.g. using name conventions) would make Event-B models more readable and easier to model communication systems.

(MG5) We have used two different ways of specifying the status register of each LUN in Section 6.6. The first is specifying as a state function and the second is specifying as state sets. Each approach is appropriate for a particular state property. Although we have found that specifying using state sets gave us a higher degree of automatic proofs, specifying using state function makes models more readable and easier to specify. Developers may choose state set if the number of state is small (e.g. two or three possible states). On the other hand, if the number of state is more than three we suggest to specify as a state function that would make the model easier to manage.

(MG6) Which direction is suitable for specifying a property? Figure 10.1 shows two possible ways of specifying a relation property between X and Y. Developers may have a difficulty of choosing which way to be addressed.

Case A is the way to specify Y as a property of X. That means we will get a property variable given below.

$$ypropOfx \in X \rightarrow Y$$

On the other hand, Case B, if we were to specify X as a property of Y, then we will have

$$xpropOfy \in Y \leftrightarrow X$$

Our guideline is to suggest that (if possible) specifying as a function is more appropriate, since the inversion can be used to obtain an inverse property. That is

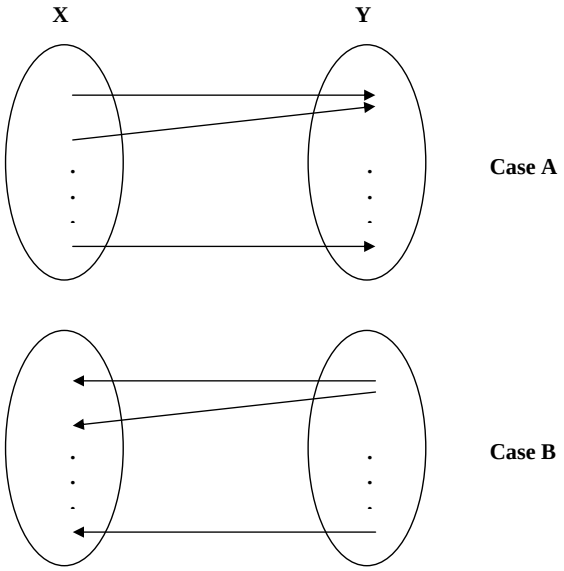


FIGURE 10.1: Two possible ways of specifying a property

$$xpropOfy = ypropOfx^{-1}$$

However, if the relation between X and Y is not a function, specifying $X \leftrightarrow Y$ or $Y \leftrightarrow X$ has no deference in modelling difficulty and proof. Thus, this depends on the users' preference. However, we have a guideline to suggest. If X is more frequently referenced than Y or there are some other properties specified like $apropOfx \in X \rightarrow A$, specifying $ypropOfx \in X \leftrightarrow Y$ would be more systematic.

(MG7) Based on our experiment in comparing two styles of specifying user-defined types: cartesian product versus projection function. In our development, these two styles have no difference in proof. However, we have found that specifying using projection function is easier to specify and follow. As example discussed in Section 6.2, we have a row address which is composed of a LUN address, a block address and a page address. If we were to specify using cartesian product as $RowAddr = LUAddr \times BAddr \times PAddr$, addressing an individual element of a row address is more complicated than using projection function. For example, getting the middle element of any row address r requires nested projection (i.e. $proj1(proj2(r))$), which is more complex than using projection function for each element.

(MG8) Redundant guards should be avoided. If any guard can be determined by another within the same event, it is not necessary to be specified. For example, suppose $wbuffer(f)$ is specified as $\mathbb{N} \nrightarrow CONTENT$, if we have $grd1 : i \notin dom(wbuffer(f))$ then $grd2 : i \mapsto cnt \notin wbuffer(f)$ is not necessary to be specified because $grd2$ is always true if $grd1$ is true.

(MG9) Conflicting guards shall be avoided. Developers should be aware of introducing a conflict guard because conflict guards are not detected and reported by the tool. For

example, we may have $grd1 : x \leq 0$ and $grd2 : x > 1$ specified as guards of an event. These two guards are conflict but have no proof showing this modelling flaw.

(MG10) To make models more readable, naming is important. Developers should be aware of naming. Namely, naming of variables, constants or even sets should be explicit and easier to follow. For example, developers may use uppercase for naming set types, while lowercase should be used as a prefix of variable names.

(MG11) As we discussed an idea of event extension proposed by Rezazadeh and Butler in [108] together with our supporting experiments presented in [44], nowadays, this idea has become an event-extension feature available in the Rodin toolset release 0.9.2 or later. The purpose of this feature is to refine a model by introducing only new properties or some extending parts to the concrete machine, as an example given in Figure 4.18. We have found that this feature is very useful for feature augmentation. It makes modelling simpler and easier to refine. In addition, 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 [45] using the Rodin tool release 0.8.2 that has no support for event-extension. We would like to suggest this feature for other developments involving horizontal refinement.

10.3 Refinement Guidelines

We presented two approaches to refine Event-B models: horizontal refinement and vertical refinement. These two approaches are based on the purpose of refinement. The horizontal is for introducing new features in refinement steps. It is also called "feature augmentation" [27]. This kind of refinement is suitable for introducing new system properties that may be postponed or missed in the abstraction. On the other hand, The purpose of the vertical refinement or structural refinement is to add more design details to the models instead of adding new features. As an example given in Chapter 4, we began with horizontal refinement steps to incrementally model an abstract file system before focusing on vertical refinement to relate the abstract file model to the flash specification. However, it is not necessary to complete all horizontal refinements before starting any vertical refinement. If we have missed some features or some properties, they can be introduced later even if some vertical refinements have been completed. For example, we have introduced the status register at the last step of refinement in Chapter 6.

Make the refinement gap as small as possible. As discussed in Chapter 4, the small gap leads to simpler gluing invariants and makes refinement not too complicated. We also believe that this help us to discover sufficient invariants. Incremental feature augmentation is effective for proof and coverage. As we have found in our experiments, this kind of development make models easier to manage and modify, and it is useful for specifying

complex systems that may have many features to be covered. To make specification easier to manage, those features can be divided into submodules and introduced in refinement steps. Refinement can also be used to introduce other requirements that may be postponed or missed from the previous steps and later be covered in the refinement steps. Refinement allows us to factor out some of the modelling and proof complexities. In our development, we began with focusing on the tree structure manipulation in the abstract model and postponed other details to other refinement steps. We did not distinguish files and directories at the abstract level. This made the proof obligations and invariants for the tree structure easier to formulate than if we had tried to model everything in one level.

10.4 Proof Guidelines

To make proof simpler and gain a higher degree of automatic proof, modelling styles, refinement techniques, etc. are important. For example, introducing proved theorems to help proof easier, sequencing of guards, careful selection of formulation, using POs as guidelines and so on, are important to achieve a higher degree of automatic proof. Some of these have already been discussed in Section 10.2.

For example, introducing additional theorems can ease proof effort. Two kinds of theorems were introduced in our development. The first type is a set of specific theorems, which were derived from the existing machine invariants and axioms. This kind of theorems can be used to specify some system properties instead of specifying as invariants. This has already been discussed in Section 10.2. The second type is a set of general theorems which are introduced and used for discharging POs. General theorems should be specified in a context. They can be seen and used by more than one machine, and can be extended by other contexts. On the other hand, specific theorems should be specified in machines. These specific theorems can be used to help discharge proof obligations as well. We did this on specific example to enforce the guidance. As already discussed in Section 4.14, we introduced some useful theorems (such as a tree-join theorem) to help proof of the tree properties. This additional theorem can be reused for discharging similar proof obligations – i.e. it was used to prove that events *create*, *copy* and *move* preserve the no-loop property. This makes interactive proof easier and can reduce the time used for proving.

Remove redundant invariants or replace some invariants by another simpler ones. For instance (as discussed in Section 6.7), we can simplify by using partition function to specify disjoint sets, instead of introducing a number of invariants saying that intersection amount of those sets is empty. This can also reduce the number of POs.

Ordering of event' guards affects proof results. In order to gain more proof effectiveness, developers must be aware of the order of guards being specified. For example, the guard

that is required to prove the well-definedness of another guard should be specified before that guard.

Use failing POs as guidelines for specifying and reasoning about system models. That is, in each step of iteration of modelling, modification and proof, POs generated by the Rodin tool could be used as guidelines to improve the model. This kind of improvement may involve removing errors and strengthening in order to help proof. For example, it can be used to determine which gluing invariant should be added to the machine or which guard should be added to the event in order to improve the model. As a result, this technique means we get a higher degree of automatic proof. The work of Ireland et al [80] proposes an automatic tool to generate modelling guidance from failed proofs. This work would be useful in the future if such guidelines are generated automatically, instead of analysing by the developers themselves that might require skilled knowledge.

Reuse the proof trees to discharge the same proof obligations, if possible. Namely, we may have some POs that have already discharged. The proof trees of these POs (maybe the whole tree or just some parts of the proof tree) can be reused to prove some POs that have the same hypothesis and goal. As an example given in the evolution of the file system model (Chapter 5), we reused proof trees of the original model to discharge the same POs of the revised model. Reuse of proof tree is done by copying the proof tree (that have already been proved) from the proof window and then pasting it onto the proof window at the same target goal. We have found that this is really useful for avoiding reproving the PO that is really complex and requires many interactive steps. Note that reusing of proof trees will not succeed if naming (of variables or parameters) of the source machine and of the target machine is different.

Chapter 11

Conclusion and Future Work

The aim of this chapter is to summarise our findings and discuss some future work. We summarise what we have carried out and achieved from each chapter in Section 11.1. An assessment of Event-B and Rodin is discussed in Section 11.2. Finally, future work is discussed in Section 11.3.

11.1 Conclusion

As previously mentioned in Chapter 1, there are many formal methods used for specification and verification. In addition, a number of useful theories and tools are available for modelling and reasoning about systems. However, they need to be improved in many ways to bridge the gap between requirements, specifications and implementations. Formal methods should be made more accessible to users. More experiments need to be carried out in order to produce scientific evidence that can convince users to deploy and gain more benefits from the use of those theories and tools. Therefore, this experimental approach was chosen as a direction of our research. Event-B and Rodin were selected as a method and a tool for our experiments. A flash-based file system was considered to be a case study of our work.

In this thesis, we have completed six main pieces of work: (i) modelling and proof of a tree-structured file system. This is later refined in Chapter 4 by focusing on read and write operations, fault-tolerance and machine decomposition; (ii) in Chapter 5, we showed an evolution of the file system model; (iii) in Chapter 6, we outlined refinement and proof of the flash interface layer which is based on the ONFI specification; (iv) in Chapter 8, we identified systematic translation of Event-B models into Java code; (v) in Chapter 9, we implemented a prototype of a flash-based file system which is a link between the first model and the second model by using translation rules of the fourth part; and (vi) in Chapter 10, we provided modelling, refinement and proof guidelines which are based on our experiments and experience of modelling.

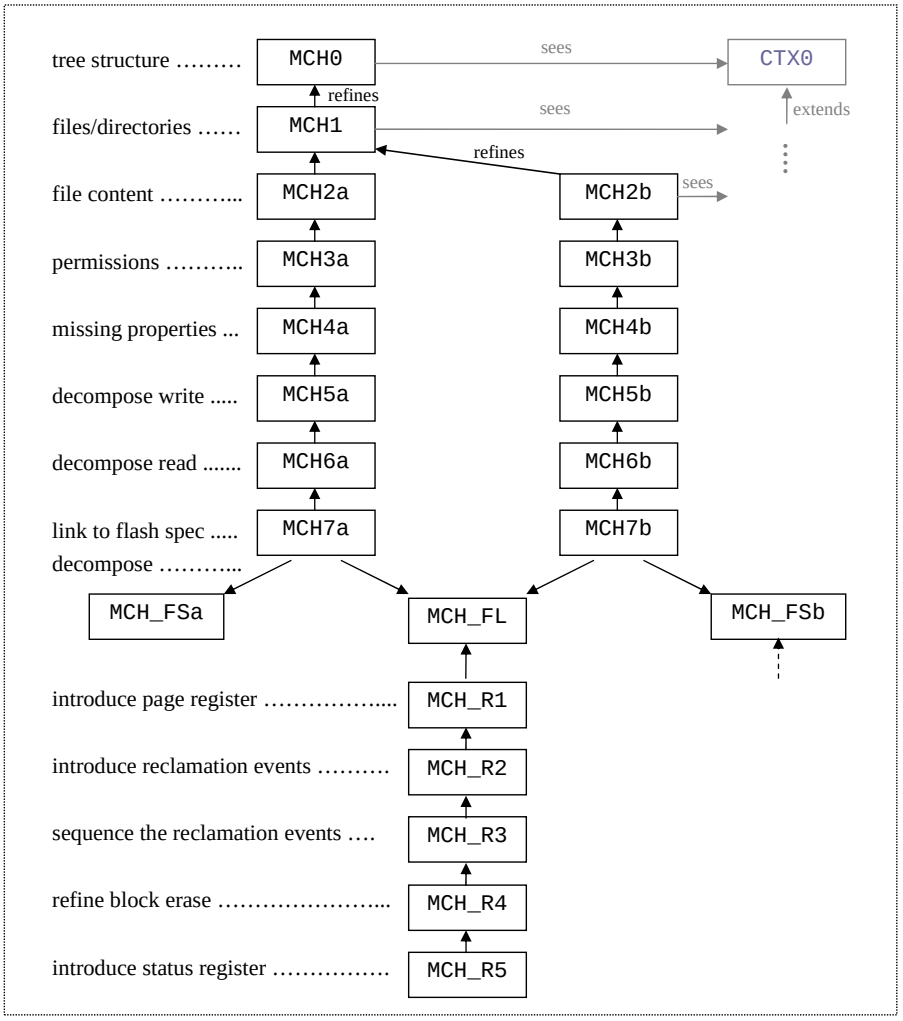


FIGURE 11.1: A diagram of refinement chains representing a development of a flash file system

Figure 11.1 shows a diagram of refinement chains representing an overview of our development of a flash-based file system. We have completed two refinement chains of the file system model (which are represented by suffixes *a* and *b*). The revised model where the requirements have been changed is represented by the chain *b*, while the original model is represented by the chain *a*. We have decomposed the file system model (MCH7x) into two sub-models representing the file system layer and the flash interface layer. After that we have explored five further refinements (MCH_R1 up to MCH_R5) focussing on the flash specification. We have got eight levels of specification (MCH0x up to MCH7x) representing the file system layer and six levels (MCH_FL up to MCH_R5) representing the flash interface layer.

In the modelling of the file system layer, we began with an abstract tree structure and later explored seven refinement steps (MCH1..MCH7) to complete the file system model. In the first step (MCH0), we focused on a tree structure and operations affecting the structure. In this step, directories and files were treated in the same way as objects in

order to make proof easier. Directories, files and other properties such as file contents, permissions were postponed to other refinements. The flash specification was introduced in the seventh refinement (MCH7x) in order to relate the abstract file system to the flash structure.

In this development, we have found some useful techniques that can ease proof effort and make models easier to manage – such as careful selection of formulation, providing additional proved theorems to help proof, use of refinement to introduce missing properties or new features; providing additional event parameters, etc. These techniques mentioned have already been proposed as guidelines and discussed in Chapter 10.

The second model represents the flash interface layer (in Chapter 6). After the machine decomposition has been used to split the file system machine into two sub-machines, we have completed five further refinement steps (MCH_R1 .. MCH_R5) focussing on the flash specification. The purpose of these refinement steps is to incrementally add more design details of the flash specification and model the wear-levelling technique. The incremental approach and other useful techniques we have found in the development of the file system model (such as selection of formulation, using POs as a guideline for correcting the model, etc.) were employed in specifying this model. Additional constraints and details were added in each step of refinements. From this model, we have completely reached 100% POs discharged automatically. We also have carried out some experiments to compare styles of modelling and proposed guidelines in Chapter 10. For example, specifying state values as state sets led us to gain higher degree of automatic proof. Specifying data types using projection function makes models easier to manage than using cartesian product, as example of specifying *RowAddr* discussed in Section 6.2.

In Chapter 7, we have discussed some related work on applying formal methods to the file store problem and provided a comparison. A distinguishing feature of our development is the use of multiple levels of refinement. In this way we relate an abstract model, with large atomic events (i.e. read and write) on abstract data structures, to a model with more complex concrete data structures, and more fine-grained atomic steps. Another distinguishing feature of our work is the use of machine decomposition to partition the system model after several refinement steps. The partitioning led to sub-models that were refined separately. As stated in [44], it is well-known that decomposition is important for scaling of formal development. However, it is rare to find examples of its application in practice. Our development of a flash file system represents an exemplar of multi-level refinement and of machine decomposition that we believe others could learn from. This acts as an exemplar and is an important contribution of our work. In addition to this contribution, in Chapter 10, we also provided some guidelines for modelling, refinement and proof that developers could learn from.

Another contribution of our work is that of providing systematic translation of Event-B models into Java code. The set of translation rules we presented in this thesis is

composed of two categories: class construction and event transformation. However, more work is required in future in order to make these rules more applicable. (Details will be discussed in Section 11.3).

The prototype we implemented was divided into two parts. One is an implementation of a flash file system which is closely related to the file system model we have specified. Another one is a simulation of flash memory which is related to the flash model in Chapter 6. This simulation part provides a number of APIs to be called by the implementation of flash file system. Our implementation follows the specification and translation rules we have developed and presented in this thesis.

The consistency of the specification is ensured by formal verification techniques, such as theorem proving and model checking. However, this cannot guarantee that the specifications/models that have been specified are the right ones or satisfy the desired requirements. Namely, if the all given requirements were not covered (maybe, because of human errors), the model/specification would not be the right one. That is, inconsistencies between the requirements and specifications may remain. This may lead developers to wrong implementations. To reduce the gap, requirements engineering techniques, including systematic translation of requirements into specification, are important. (But this is out off our research scope.) In our development, to make translation of requirements into specifications more systematic, requirements in each step were listed as bullet points of required properties (as presented in Chapter 4). Some properties (e.g. no-loop property) were then expressed as invariants that need to be held forever while some properties/requirements were expressed as events. Namely, whenever the state variables have been changed (by events specified) the associated invariants much be proved to be maintained. If all required properties/requirements could be related to/explained by invariants or any part of the model, then it would be more confident to guarantee that the model specified is correct/valid.

11.2 Assessment of Event-B and the Rodin tool

The purpose of this section is to assess Event-B and Rodin, which are used in our development for specifying and reasoning about the flash file system. The assessments discussed below are based on our experiments and experiences of using Event-B and the Rodin platform.

11.2.1 Event-B

In this section, we begin with outlining positive points of Event-B and then providing some guidelines and desirable features that may be useful in the feature.

The structure that obviously separates machines from contexts makes an Event-B model easier to refine. Namely, the machines and the contexts can be individually refined or referenced by others. Additionally, the flexibility of refinement in Event-B allows users to decide which approach or technique they want to employ. For example, in case of an incremental approach, users can postpone some requirements at the beginning and later address them in other refinement steps.

In addition, even if the specification goes wrong or there is something that cannot be proved, it is possible to check where and why it is incorrect by using generated proof obligations as a guideline. For example, each type of the proof obligations, such as GRD (guard), INV (invariant), etc., can be used to tell what the system is trying to prove and why it cannot be discharged.

As already discussed in this report, many useful features have been added to the language, such as partition operation, event-extension, etc. These features make modelling and proof simpler. The event-extension is very useful for horizontal refinement. This makes models easier to be refined and proved. In addition, some modifications can easily be made to the abstraction individually and are propagated down automatically. The partition operation makes modelling of state sets simpler. We do not need to specify a huge number of invariants to clarify that the state sets are disjoint. However, some features and theories that may be useful for specification should be added to the language and tools such as providing useful theories, separation of internal and external parameters, sequence type, etc.

Distinguishing of parameters between input and output would be useful for specifying interactive systems. The distinction makes users easier to know which parameters should be passed to the event and which parameters are local to that event.

Providing a *sequence* type and its manipulation (like supporting in the B-method) would be useful for specifying the models that require this type in Event-B. For example, if we were to specify the structure of a file system as a naming-structure, using sequence type would make the model easier to manage. Similarly for the transitive closure, providing it would be useful for modelling as well. Developers could use it directly, instead of specifying by themselves.

Based on the *IsTree* predicate we have introduced in the Proof section of Chapter 4, we believe that providing tree theories as a feature of the language might be useful for specifying systems involving manipulation of the tree structure. Tree operations *join*, *split* and *duplicate* can be defined as a theory used for manipulating a tree. For example, copying a subtree from one place to another can be done by *duplicating* the given subtree and then *joining* the replica with the node of the target tree specified. In order to remove a subtree, *split* the subtree specified and then discard it.

11.2.2 Rodin

Rodin is not only a tool for specification but also a tool for refinement and verification. Rodin comes with a database of modelling elements and useful plug-ins such as a proof obligation generator (POG), model checkers, automated and interactive provers [7]. The automatic PO generator and provers are useful features for verification that can save users' time from manual proving. For example, in case of Z/EVES (a proof tool for Z, but have no POG), users need to identify POs by themselves. In addition, without automatic provers, users have to know what goal they need to prove and which hypothesis should be used to prove that goal. Even though the users are good at proving, they may have to spend a lot of time to discharge a huge number of proof obligations by themselves. In case of the Rodin tool, as already discussed in Section 3.9, proof obligations are automatically generated and then are proved automatically by the provers where possible. Although some POs are not discharged automatically, the tool provides an interactive prover which is easy to use. However, the prover needs to be improved in some ways. For example, the memory problem that always occur when proving some complex POs should be solved. Another case, some trivial POs that should be discharged automatically still need to be proved interactively. Sometimes, some necessary hypothesis used to prove the given goals are missed (are not selected automatically by the tool). Occasionally, many hypothesis which are not necessary are added to the list of hypothesis used for automatic proof. These examples may mean the POs cannot be discharged automatically.

Additionally, we now consider our experiences of using the Rodin platform as a tool for constructing and analysing Event-B models. The Rodin platform provides a useful tool for Event-B specification. It uses a visual interface which is familiar to users as in other modern software. Each component used for specification is well-designed and easy to use. In addition, a refinement can be constructed easily by this tool.

Moreover, as an extensible tool which allows users to customise their tool and plug in other available tools to satisfy their needs, it makes this tool more flexible and attractive to use. For instance, the users can install UML-B plug-in to design their models using components provided by UML-B; plug in a decomposition tool to decompose a machine; or plug in B2Latex as a tool to generate latex documents, etc.

In order to satisfy additional features of the language proposed in the previous section, the Rodin tool should be adapted or additional plug-ins should be provided. For instance, providing code generators are important for bridging the gap between the specification and the implementation. It would be good to develop a code generator to translate Event-B models into programming code such as Java, C, etc.

It would be useful, if conflicting guards could be detected and reported by the tool. As discussed in Section 10.2, currently, conflicting guards can make models go wrong without reporting any errors. Thus, it is really important to be aware of this situation.

Fortunately, those useful features, such as a decomposition plug-in, and code generator are considered as roadmap features, and are being developed for the Rodin toolset [7].

Finally, although the Rodin tool supports the event-extension feature that is useful for feature augmentation, some improvements should be made. For instance, while extending an event, the previous abstract specification should be shown in the editor as a disabled part. This would make it easier for developers to follow what they have specified and what they are trying to extend, instead of hiding it in the editor.

11.3 Future Work

Based on our development, we have seen some issues/features that could be explored in the future in order to push forward research in formal methods. For example, developing diagrammatic forms of guidance, comparison of decomposition styles, tools supporting the generation of useful lemmas, verification of translation, etc. These would be useful and make formal methods more accessible in the future.

Firstly, it would be useful in future if diagrammatic forms of guidance – such as event-refinement diagrams (e.g. Figure 4.23, Figure 6.1, etc.) and refinement-chain diagrams (e.g. Figure 11.1 where we represent the overview of the specification process) could be developed as plug-ins. We believe these diagrams help us to understand more about the system being specified. To make it more formal, not just an aid that need to be drawn by hand, investigation for more formal incorporation of these diagrams into refinement proof could be carried out, together with tools supporting these features.

Secondly, based on our use of shared event decomposition, we see an issue that would be useful in the future. Since Rodin provides two types of decomposition (shared variable and shared event, as discussed in Section 3.6), and a decomposition tool is now available and being improved, carrying out an experiment of comparison of these two approaches would be useful to provide scientific evidence for developers/modellers. For example, what the strengths and the weaknesses of these two approaches are; what kind of systems they are suitable for, etc.

Thirdly, based on our experience of introducing proved theorems/lemmas to help proofs, it would be useful if there were tools/plug-ins for generation of lemmas. For example, as discussed in Section 4.14, we have introduced some proved theorems to help proof of tree properties (e.g. no-loop property). Instead of discovering and introducing the lemmas by the modellers themselves, providing modellers with a tool to generate lemmas/theorems would be better.

Fourthly, providing a tool to automatically generate proof guidance from the failing proof obligations is another challenge to be addressed. Regarding the proof guideline we discussed in Section 10.4, we can used failed proof as guidelines to improve (correcting

or strengthening) the models. However, use of failing proofs requires skilled knowledge in formal reasoning. Hence, developing a tool that can generate a list of guidelines from those failed proof obligations would help developers a lot in modelling and proof.

Finally, as discussed in Chapter 8, our translation rules do not cover all possible forms of Event-B notation, such as relation operations (e.g. domain/range subtraction, overriding, etc.). Additional patterns and systematic rules for translation of set and relation operations are still required. In addition, similar to BART [107] that have already discussed in Section 8.4, it would be useful to have an automatic tool for systematic application of the translation rules. For example, an automatic tool for refining Event-B models into the normal forms that are able to be translated using the rules. Moreover, formal verification of the translation is also required to ensure the correctness of the translation. These would be useful in future to make the translation rules more applicable.

Appendix A

An Event-B specification of a file system

A.1 An initial model: Tree structure

Tree structured model of file store. There is a single root object. Each object other than parent has a root. There are no loops in the parent structure. Each object is reachable from the root.

MACHINE FMCH00
SEES FCTX01
VARIABLES

`objects`
`parent`

INVARIANTS

`inv1` : $objects \in \mathbb{P}(OBJECT)$
`inv2` : $root \in objects$
`inv3` : $parent \in objects \setminus \{root\} \rightarrow objects$
`inv4` : $\forall s \cdot (s \subseteq parent^{-1}[s] \Rightarrow s = \varnothing)$

No loop: easier to discharge POs than using transitive closure

`thm1` : $tcl(parent) \cap (OBJECT \lhd id) = \varnothing$

No loop property using transitive closure

`thm2` : $\forall T \cdot root \in T \land parent^{-1}[T] \subseteq T \Rightarrow objects \subseteq T$

used to prove thm3

`thm3` : $objects \subseteq \{root\} \cup (tcl(parent))^{-1}[\{root\}]$

used to prove thm4

`thm4` : $(objects \setminus \{root\}) \subseteq (tcl(parent))^{-1}[\{root\}]$

Reachabilty property, all objects can be reached from the root node

`thm5` : $\forall x \cdot x \notin ran(parent) \Rightarrow (tcl(parent))^{-1}[\{x\}] = \varnothing$

All leaf nodes have no children

thm6 : $\forall x \cdot (tcl(parent))^{-1}[\{x\}] \lhd parent \in (tcl(parent))^{-1}[\{x\}] \to (tcl(parent))^{-1}[\{x\}] \cup \{x\}$

thm7 : $\forall x \cdot (tcl(parent))^{-1}[\{x\}] \lhd parent \in ((tcl(parent))^{-1}[\{x\}] \cup \{x\}) \setminus \{x\} \to (tcl(parent))^{-1}[\{x\}] \cup \{x\}$

this is used for copying, inserting and moving in order to prove that any subtree rooted at x is a total function.

thm8 : $\forall x, s \cdot s \subseteq ((tcl(parent))^{-1}[\{x\}] \lhd parent)^{-1}[s] \Rightarrow s = \varnothing$

thm7 plus thm8 implies that any subtree rooted at x is a tree

thm9 : $\forall x \cdot ((tcl(parent))^{-1}[\{x\}] \cup \{x\}) \lhd\!\!\!- parent \in (objects \setminus ((tcl(parent))^{-1}[\{x\}] \cup \{x\})) \setminus \{root\} \to objects \setminus ((tcl(parent))^{-1}[\{x\}] \cup \{x\})$

this is used for delete and move operation in order to prove inv3.

EVENTS

Initialisation

begin

act1 : $objects := \{root\}$

act4 : $parent := \varnothing$

end

Event *newobj* $\widehat{=}$

This event creates a new object (*obj*) with a specified parent (*indr*).

any

$obj, indr$

where

grd1 : $obj \in OBJECT \setminus objects$

grd2 : $indr \in objects$

then

act1 : $objects := objects \cup \{obj\}$

act2 : $parent(obj) := indr$

end

Event *move* $\widehat{=}$

Move an object *obj* to another place *to*.

any

obj

to

des all descendants of *obj*

where

grd1 : $obj \in objects \setminus \{root\}$

grd2 : $to \in objects$

grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$

grd4 : $to \notin des \cup \{obj\}$

then

act1 : $parent(obj) := to$

end

Event *delete* $\widehat{=}$

Delete an object that has no children.

any

obj

where

grd1 : $obj \in objects \setminus \{root\}$

grd2 : $parent^{-1}[\{obj\}] = \varnothing$

then

act1 : $objects := objects \setminus \{obj\}$

act2 : $parent := \{obj\} \lhd parent$

end

Event *copy* $\widehat{=}$

Copy an object *obj* and all its descendants (*des*) to another location (*to*).

any

obj, des, to

$objs$ all objects to be copied

$corres$ corresponding function mapping source objects to their copies

$nobjs$ new copies of *objs*

$nobj$ the copy of *obj*

$subparent$ subtree to be copied

$replica$ the copy of *subparent*

where

grd1 : $obj \in objects \setminus \{root\}$

grd2 : $des \subseteq objects$

grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$

grd4 : $to \in objects$

grd5 : $to \notin des \cup \{obj\}$

grd6 : $objs = des \cup \{obj\}$

grd7 : $nobjs \subseteq OBJECT \setminus objects$

grd8 : $corres \in objs \rightarrowtail\!\!\!\!\!\rightarrow nobjs$

grd9 : $nobj = corres(obj)$

grd10 : $subparent = des \lhd parent$

grd11 : $replica = corres^{-1};\ subparent;\ corres$

then

act1 : $parent := parent \cup replica \cup \{nobj \mapsto to\}$

act2 : $objects := objects \cup nobjs$

end

Event *deltree* $\widehat{=}$

Delete the given object *obj* and all its descendants (*des*).

any

$obj, des, objs$

where

grd1 : $obj \in objects \setminus \{root\}$

grd2 : $des \subseteq objects$

grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$

grd4 : $objs = des \cup \{obj\}$

then

act1 : $objects := objects \setminus objs$

act2 : $parent := objs \lhd parent$

end

Event *mount* $\widehat{=}$

Mount a flash device into an existing root.

any

$objs$ all objects to be mounted

prt parent structure of *objs* rooted at x

x subroot

where

grd1 : $objs \subseteq OBJECT$

grd2 : $x \in objs$

grd3 : $prt \in objs \setminus \{x\} \rightarrow objs$

grd4 : $\forall s \cdot (s \subseteq prt^{-1}[s] \Rightarrow s = \varnothing)$

Has no loops.

grd5 : $prt \cap parent = \varnothing$

grd6 : $objects \cap objs = \varnothing$

then

act1 : $objects := objects \cup objs$

act2 : $parent := parent \cup prt \cup \{x \mapsto root\}$

end

Event *unmount* $\widehat{=}$

any

objs all objects to be released

x subroot to be unmounted.

where

grd1 : $objs \subseteq objects$

grd2 : $root \notin objs$

grd3 : $x \in objs$

grd4 : $objs = (tcl(parent))^{-1}[\{x\}] \cup \{x\}$

then

act1 : $objects := objects \setminus objs$

act2 : $parent := objs \lhd parent$

end

END

A.2 The first refinement: Files and directories

In this refinement, *objects* are partitioned into *files* and *directories*. *root* is a directory. Any parent is a directory. Variable *objects* is no longer used.

MACHINE FMCH01
REFINES FMCH00
SEES FCTX01
VARIABLES
$files$
$directories$
`parent`
INVARIANTS
`inv1` : $files \subseteq objects$
`inv2` : $directories \subseteq objects$
`inv3` : $files \cap directories = \varnothing$
`inv4` : $objects = files \cup directories$
`inv5` : $root \in directories$
`inv6` : $ran(parent) \subseteq directories$
EVENTS
Initialisation
begin
`act2` : $files := \varnothing$
`act3` : $directories := \{root\}$
`act4` : $parent := \varnothing$
end
Event *mkdir* $\widehat{=}$
Make a directory *obj* in the given directory *indr*.
refines *newobj*
any
$obj, indr$
where
`grd1` : $obj \in OBJECT \setminus (files \cup directories)$
`grd2` : $indr \in directories$
then
`act1` : $directories := directories \cup \{obj\}$
`act2` : $parent(obj) := indr$
end
Event *crt_file* $\widehat{=}$
Create file *obj* in the given directory *indr*
refines *newobj*
any
$obj, indr$
where
`grd1` : $obj \in OBJECT \setminus (files \cup directories)$
`grd2` : $indr \in directories$
then
`act1` : $files := files \cup \{obj\}$

act2 : $parent(obj) := indr$

end

Event *move* $\widehat{=}$

Move an object *obj* and its descendants *des* to another place *to*.

refines *move*

any

obj, to, des

where

grd1 : $obj \in (files \cup directories) \setminus \{root\}$
grd2 : $to \in directories$
grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$
grd4 : $to \notin des \cup \{obj\}$

then

act1 : $parent(obj) := to$

end

Event *delfile* $\widehat{=}$

Delete a file (*obj*)

refines *delete*

any

obj

where

grd1 : $obj \in files$
grd2 : $parent^{-1}[\{obj\}] = \varnothing$

then

act1 : $files := files \setminus \{obj\}$
act2 : $parent := \{obj\} \lhd parent$

end

Event *rmdir* $\widehat{=}$

Delete an empty directory.

refines *delete*

any

obj

where

grd1 : $obj \in directories \setminus \{root\}$
grd2 : $parent^{-1}[\{obj\}] = \varnothing$

then

act1 : $directories := directories \setminus \{obj\}$
act2 : $parent := \{obj\} \lhd parent$

end

Event *copy* $\widehat{=}$

Copy an existing object *obj* all its descendants *des* to another location *to*

refines *copy*

any

$obj, des, to, objs, corres$
$nobjs, nobj, subparent, replica$

where

grd1 : $obj \in (files \cup directories) \setminus \{root\}$
grd2 : $des \subseteq (files \cup directories)$
grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$
grd4 : $to \in directories$
grd5 : $to \notin des \cup \{obj\}$
grd6 : $objs = des \cup \{obj\}$
grd7 : $nobjs \subseteq OBJECT \setminus (files \cup directories)$

grd8 : $corres \in objs \rightarrowtail\!\!\!\!\!\!\rightarrow nobjs$
grd9 : $nobj = corres(obj)$
grd10 : $subparent = des \lhd parent$
grd11 : $replica = corres^{-1};\ subparent;\ corres$

then

act1 : $parent := parent \cup replica \cup \{nobj \mapsto to\}$
act2 : $files := files \cup corres[objs \cap files]$
act3 : $directories := directories \cup corres[objs \cap directories]$

end

Event *deltree* $\widehat{=}$

Delete the given object (*obj*) and all its descendants (*des*). Actually, it can be done recursively by events delfile and rmdir.

refines *deltree*

any

$obj, des, objs$

where

grd1 : $obj \in (files \cup directories) \setminus \{root\}$
grd2 : $des \subseteq (files \cup directories)$
grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$
grd4 : $objs = des \cup \{obj\}$

then

act1 : $parent := objs \lhd\!\!\!\!- parent$
act2 : $files := files \setminus (objs \cap files)$
act3 : $directories := directories \setminus (objs \cap directories)$

end

Event *mount* $\widehat{=}$

refines *mount*

any

$objs, fs, ds, prt, x$

fs: set of files, *ds* set of directories

where

grd1 : $objs \subseteq OBJECT$
grd2 : $fs \subseteq objs$
grd3 : $ds \subseteq objs$
grd4 : $objs = fs \cup ds$
grd5 : $fs \cap ds = \varnothing$
grd6 : $(files \cup directories) \cap objs = \varnothing$
grd7 : $x \in ds$
grd8 : $prt \in (fs \cup ds) \setminus \{x\} \rightarrow ds$
grd9 : $\forall s \cdot (s \subseteq prt^{-1}[s] \Rightarrow s = \varnothing)$
grd10 : $prt \cap parent = \varnothing$
grd11 : $files \cap fs = \varnothing$
grd12 : $directories \cap ds = \varnothing$

then

act1 : $files := files \cup fs$
act2 : $directories := directories \cup ds$
act3 : $parent := parent \cup prt \cup \{x \mapsto root\}$

end

Event *unmount* $\widehat{=}$

refines *unmount*

Unmount the storage device. All objects *objs* rooted at x within the device will be released.

any

$objs, x$

where

grd1 : $objs \subseteq files \cup directories$

grd2 : $root \notin objs$

grd3 : $x \in objs$

grd4 : $objs = (tcl(parent))^{-1}[\{x\}] \cup \{x\}$

then

act1 : $files := files \setminus (objs \cap files)$

act2 : $directories := directories \setminus (objs \cap directories)$

act3 : $parent := objs \lhd parent$

end

END

A.3 The second refinement: File content

Introduce file content together with open, read and write events. Power loss and power on are also introduced in this refinement.

MACHINE FMCH02
REFINES FMCH01
SEES FCTX02
VARIABLES

`files`
`directories`
`parent`
$fcontent$ the content of each file
w_opened_files files which are opened for writing
r_opened_files files which are opened for reading
$wbuffer$ write buffers
$rbuffer$ read buffers
$power\_on$ power status

INVARIANTS

inv1 : $fcontent \in files \rightarrow CONTENT$
inv2 : $w\_opened\_files \subseteq files$
inv3 : $r\_opened\_files \subseteq files$
inv4 : $w\_opened\_files \cap r\_opened\_files = \varnothing$
inv5 : $wbuffer \in w\_opened\_files \rightarrow CONTENT$
inv6 : $rbuffer \in r\_opened\_files \rightarrow CONTENT$
inv7 : $power\_on \in BOOL$
inv8 : $power\_on = FALSE \Rightarrow (w\_opened\_files := \varnothing \land r\_opened\_files := \varnothing$
$wbuffer := \varnothing \land rbuffer := \varnothing)$

EVENTS
Initialisation

extended

begin
act4 : $fcontent := \varnothing$
act5 : $w\_opened\_files := \varnothing$
act6 : $r\_opened\_files := \varnothing$
act7 : $wbuffer := \varnothing$
act8 : $rbuffer := \varnothing$
act9 : $power\_on := TRUE$
end

Event $mkdir \mathrel{\widehat{=}}$
extends *mkdir*
where
grd3 : $power\_on = TRUE$
end

Event $crt\_file \mathrel{\widehat{=}}$
Create a file *obj*

extends *crt_file*

where
grd3 : $power\_on = TRUE$
then
act3 : $fcontent(obj) := \varnothing$
end

Event *move* $\widehat{=}$

Move an object from one place to another.

extends *move*

where
grd5 : $power\_on = TRUE$
grd6 : $obj \notin w\_opened\_files \cup r\_opened\_files$
end

Event *delfile* $\widehat{=}$

Delete a file from the specified directory.

extends *delfile*

where
grd3 : $obj \notin w\_opened\_files \cup r\_opened\_files$
grd4 : $power\_on = TRUE$
then
act3 : $fcontent := \{obj\} \lhd fcontent$
end

Event *rmdir* $\widehat{=}$

Delete an empty directory

extends *rmdir*

where
grd3 : $power\_on = TRUE$
end

Event *deltree* $\widehat{=}$

Delete an object and its descendants.

extends *deltree*

where
grd5 : $objs \cap (w\_opened\_files \cup r\_opened\_files) = \varnothing$
All objects to be deleted, *objs*, must not be in used.
grd6 : $power\_on = TRUE$
then
act4 : $fcontent := objs \lhd fcontent$
end

Event *copy* $\widehat{=}$

Copy an existing object

extends *copy*

where
grd12 : $powerloss = FALSE$
then
act4 : $fcontent := fcontent \cup (corres^{-1}; fcontent)$
end

Event *w_open* $\widehat{=}$

Open the given file f for writing.

any

f, cnt

where

grd1 : $f \in files$

grd2 : $cnt \in CONTENT$

The content to be written.

grd3 : $f \notin w\_opened\_files \cup r\_opened\_files$

grd4 : $power\_on = TRUE$

then

act1 : $w\_opened\_files := w\_opened\_files \cup \{f\}$

act2 : $wbuffer(f) := cnt$

Set *wbuffer* pointing to the content *cnt* to be written.

end

Event *r_open* $\widehat{=}$

Open file f for reading.

any

f

where

grd1 : $f \in files$

grd2 : $f \notin w\_opened\_files \cup r\_opened\_files$

grd3 : $power\_on = TRUE$

then

act1 : $r\_opened\_files := r\_opened\_files \cup \{f\}$

act2 : $rbuffer(f) := \varnothing$

end

Event *readfile* $\widehat{=}$

Read the whole content of a file from the storage into the read buffer.

any

f

where

grd1 : $f \in r\_opened\_files$

grd2 : $power\_on = TRUE$

then

act1 : $rbuffer(f) := fcontent(f)$

end

Event *writefile* $\widehat{=}$

Write the content on the write buffer of the given file into the storage.

any

f

where

grd1 : $f \in w\_opened\_files$

grd2 : $power\_on = TRUE$

then

act1 : $fcontent(f) := wbuffer(f)$

end

Event *close* $\widehat{=}$

Close an opened file.

any

f

where

grd1 : $f \in r\_opened\_files \cup w\_opened\_files$

grd2 : $power\_on = TRUE$

then

act1 : $r\_opened\_files := r\_opened\_files \setminus \{f\}$

act2 : $w\_opened\_files := w\_opened\_files \setminus \{f\}$

act3 : $rbuffer := \{f\} \lhd rbuffer$
act4 : $wbuffer := \{f\} \lhd wbuffer$
end

Event *power_off* $\widehat{=}$
when
grd1 : $power\_on = TRUE$
then
act1 : $power\_on := FALSE$
act2 : $wbuffer := \varnothing$
act3 : $rbuffer := \varnothing$
act4 : $w\_opened\_files := \varnothing$
act5 : $r\_opened\_files := \varnothing$
end

Event *power_on* $\widehat{=}$
when
grd1 : $power\_on = FALSE$
then
act1 : $power\_on := TRUE$
end

Event *mount* $\widehat{=}$
Close the device into the existing root.
extends *mount*
any
$fcnt$
where
grd13 : $fcnt \in fs \rightarrow CONTENT$
grd14 : $power\_on = TRUE$
then
act4 : $fcontent := fcontent \cup fcnt$
end

Event *unmount* $\widehat{=}$
extends *unmount*
where
grd5 : $objs \cap w\_opened\_files = \varnothing$
grd6 : $objs \cap r\_opened\_files = \varnothing$
grd7 : $power\_on = TRUE$
then
act4 : $fcontent := objs \lhd fcontent$
end

END

A.4 The third refinement: Permissions

Introduce permissions and related events.

MACHINE FMCH03

VARIABLES

...
users the set of existing users
groups the set of existing groups
user_grps user's groups
user_pgrp the primary group of each user
obj_owner the owner of each object
obj_grp the group-owner of each object
obj_perms permissions of each object

INVARIANTS

inv1 : $users \subseteq USER$
inv2 : $groups \subseteq GROUP$
inv3 : $su \in users$
inv4 : $admin \in groups$
inv5 : $user\_grps \in users \leftrightarrow groups$
inv6 : $user\_pgrp \in users \rightarrow groups$
inv7 : $obj\_owner \in (files \cup directories) \rightarrow users$
inv8 : $obj\_grp \in (files \cup directories) \rightarrow groups$
inv9 : $obj\_perms \in (files \cup directories) \leftrightarrow PERMISSION$
thm1 : $obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps \in dom(WPerm)$
thm2 : $obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps \in dom(RPerm)$
thm3 : $obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps \in dom(XPerm)$

EVENTS

Initialisation

extended

begin
act10 : $users := \{su\}$
act11 : $groups := \{admin\}$
act12 : $user\_grps := \{su \mapsto admin\}$
act13 : $user\_pgrp := \{su \mapsto admin\}$
act14 : $obj\_owner := \{root \mapsto su\}$
act15 : $obj\_grp := \{root \mapsto admin\}$
act16 : $obj\_perms := \{root \mapsto wbo, root \mapsto rbo, root \mapsto xbo\}$
end

Event *mkdir* $\widehat{=}$

Make a directory

extends *mkdir*

any
usr the user who issues the request
grp the primary group of the user
where
grd4 : $usr \in users$

grd5 : $grp \in groups$
grd6 : $usr \mapsto grp \in user\_pgrp$
grd7 : $indr \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act3 : $obj\_owner(obj) := usr$
act4 : $obj\_grp(obj) := grp$
act5 : $obj\_perms := obj\_perms \cup \{obj \mapsto rbo, obj \mapsto wbo, obj \mapsto xbo\}$

end

Event *crt_file* $\widehat{=}$

Create a file

extends *crt_file*

any

usr the user who issues the request
grp the primary group of the user

where

grd4 : $usr \in users$
grd5 : $grp \in groups$
grd6 : $usr \mapsto grp \in user\_pgrp$
grd7 : $indr \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act4 : $obj\_owner(obj) := usr$
act5 : $obj\_grp(obj) := grp$
act6 : $obj\_perms := obj\_perms \cup \{obj \mapsto rbo, obj \mapsto wbo, obj \mapsto xbo\}$

end

Event *move* $\widehat{=}$

Move an object from one place to another.

extends *move*

any

usr

where

grd7 : $usr \in users$
grd8 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
grd9 : $to \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

end

Event *delfile* $\widehat{=}$

Delete file *obj* by user *usr*

extends *delfile*

any

usr

where

grd5 : $usr \in users$
grd6 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act4 : $obj\_owner := \{obj\} \lhd obj\_owner$
act5 : $obj\_grp := \{obj\} \lhd obj\_grp$
act6 : $obj\_perms := \{obj\} \lhd obj\_perms$

end

Event *rmdir* $\widehat{=}$

Delete an empty directory (*obj*) by user *usr*.

extends *rmdir*

any

usr

where

grd4 : $usr \in users$
grd5 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act3 : $obj\_owner := \{obj\} \lhd obj\_owner$
act4 : $obj\_grp := \{obj\} \lhd obj\_grp$
act5 : $obj\_perms := \{obj\} \lhd obj\_perms$

end

Event *deltree* $\widehat{=}$

Delete the given object and all its descendants.

extends *deltree*

any

usr

where

grd7 : $usr \in users$
grd8 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act5 : $obj\_owner := objs \lhd obj\_owner$
act6 : $obj\_grp := objs \lhd obj\_grp$
act7 : $obj\_perms := objs \lhd obj\_perms$

end

Event *copy* $\widehat{=}$

Copy an existing object *obj* to directory *to* by user *usr*

extends *copy*

any

usr

where

grd13 : $usr \in users$
grd14 : $obj \mapsto usr \in RPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
grd15 : $to \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

then

act5 : $obj\_owner := obj\_owner \cup (corres^{-1};\ obj\_owner)$
act6 : $obj\_grp := obj\_grp \cup (corres^{-1};\ obj\_grp)$
act7 : $obj\_perms := obj\_perms \cup (corres^{-1};\ obj\_perms)$

end

Event *w_open* $\widehat{=}$

Open file f for writing by user *usr*.

extends *w_open*

any

usr

where

grd5 : $usr \in users$
grd6 : $f \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

end

Event *r_open* $\widehat{=}$

Open file f for reading by user *usr*

extends *r_open*

any

usr

where

grd4 : $usr \in users$
grd5 : $f \mapsto usr \in RPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$

end

Event *readfile* $\widehat{=}$

Read the whole content of a file from the storage into the read buffer.

extends *readfile*

Event *writefile* $\widehat{=}$

Write the content on the wbuffer of the given file into the storage.

extends *writefile*

Event *close* $\widehat{=}$

Close an opened file.

extends *close*

Event *power_off* $\widehat{=}$

extends *power_off*

Event *power_on* $\widehat{=}$

extends *power_on*

Event *mount* $\widehat{=}$

extends *mount*

any

objown the owner of each object being mounted
objperms list of permissions of each object
objgrp the group owner of each object

where

grd15 : $objown \in objs \rightarrow users$
grd16 : $objperms \in objs \leftrightarrow PERMISSION$
grd17 : $objgrp \in objs \rightarrow groups$

then

act5 : $obj\_owner := obj\_owner \cup objown$
act6 : $obj\_perms := obj\_perms \cup objperms$
act7 : $obj\_grp := obj\_grp \cup objgrp$

end

Event *unmount* $\widehat{=}$

extends *unmount*

then

act5 : $obj\_owner := objs \lhd obj\_owner$
act6 : $obj\_grp := objs \lhd obj\_grp$
act7 : $obj\_perms := objs \lhd obj\_perms$

end

END

A.5 The fourth refinement: Missing properties

Adding other properties: name, creation date and last modification date.

MACHINE FMCH04
REFINES FMCH03
SEES FCTX03
VARIABLES

...
oname name of each object
dateCreated creation date
dateLastModified last modification date
$file\_size$ file size

INVARIANTS

inv1 : $oname \in (files \cup directories) \rightarrow NAME$
inv2 : $dateCreated \in (files \cup directories) \rightarrow DATE$
inv3 : $dateLastModified \in (files \cup directories) \rightarrow DATE$
inv4 : $file\_size \in files \rightarrow \mathbb{N}$
thm1 : $files \cap directories = \varnothing$

EVENTS
Initialisation

extended

begin
act17 : $oname := \{root \mapsto rname\}$
act18 : $dateCreated := \{root \mapsto dfdate\}$
act19 : $dateLastModified := \{root \mapsto dfdate\}$
act20 : $file\_size := \varnothing$
end

Event *mkdir* $\widehat{=}$

Make a directory

extends *mkdir*

any
nme
where
grd7 : $nme \in NAME$
grd8 : $nme \notin oname[parent^{-1}[\{indr\}]]$
then
act6 : $oname(obj) := nme$
act7 : $dateCreated(obj) := nowdate$
act8 : $dateLastModified(obj) := nowdate$
end

Event *crt_file* $\widehat{=}$

create file

extends *crt_file*

any
nme
where

grd7 : $nme \in NAME$
grd8 : $nme \notin oname[parent^{-1}[\{indr\}]]$

then

act7 : $oname(obj) := nme$
act8 : $dateCreated(obj) := nowdate$
act9 : $dateLastModified(obj) := nowdate$
act10 : $file\_size(obj) := 0$

end

Event *move* $\widehat{=}$

Move an object from one place to another.

extends *move*

where

grd10 : $oname(obj) \notin oname[parent^{-1}[\{to\}] \cup \{to\}]$

end

Event *delfile* $\widehat{=}$

Delete a file *obj*

extends *delfile*

then

act9 : $oname := \{obj\} \lhd oname$
act10 : $dateCreated := \{obj\} \lhd dateCreated$
act11 : $dateLastModified := \{obj\} \lhd dateLastModified$
act12 : $file\_size := \{obj\} \lhd file\_size$

end

Event *rmdir* $\widehat{=}$

Delete an empty directory *obj*

extends *rmdir*

then

act6 : $oname := \{obj\} \lhd oname$
act7 : $dateCreated := \{obj\} \lhd dateCreated$
act8 : $dateLastModified := \{obj\} \lhd dateLastModified$

end

Event *deltree* $\widehat{=}$

Delete the given object and all its descendants.

extends *deltree*

then

act8 : $oname := objs \lhd oname$
act9 : $dateCreated := objs \lhd dateCreated$
act10 : $dateLastModified := objs \lhd dateLastModified$
act11 : $file\_size := objs \lhd file\_size$

end

Event *copy* $\widehat{=}$

Copy an existing object *obj* to directory *to*.

extends *copy*

where

grd17 : $oname(obj) \notin oname[parent^{-1}[\{to\}] \cup \{to\}]$

then

act9 : $oname := oname \cup (corres^{-1};\ oname)$
act10 : $dateCreated := dateCreated \cup (corres^{-1};\ dateCreated)$
act11 : $dateLastModified := dateLastModified \cup (corres^{-1};\ dateLastModified)$
act12 : $file\_size := file\_size \cup (corres^{-1};\ file\_size)$

end

Event *w_open* $\widehat{=}$

Open the given file f for writing.

extends *w_open*

Event *r_open* $\widehat{=}$

Open the given file for reading.

extends *r_open*

Event *readfile* $\widehat{=}$

Read the whole content of a file from the storage into the read buffer.

extends *readfile*

Event *writefile* $\widehat{=}$

Write the content on the *wbuffer* of the given file into the storage.

extends *writefile*

then

act2 : $dateLastModified(f) := nowdate$
act3 : $file\_size(f) := card(wbuffer(f))$

end

Event *close* $\widehat{=}$

Close an opened file.

extends *close*

Event *rename* $\widehat{=}$

any

obj the given object to be renamed
$indr$ the directory to which the given object belongs
$nname$ new name

where

grd1 : $obj \in (files \cup directories) \setminus \{root\}$
grd2 : $obj \notin (w\_opened\_files \cup r\_opened\_files)$
grd3 : $indr \in directories$
grd4 : $nname \in NAME$
grd5 : $indr = parent(obj)$
grd6 : $nname \notin oname[parent^{-1}[\{indr\}] \cup \{indr\}]$

then

act1 : $oname(obj) := nname$
act2 : $dateLastModified(obj) := nowdate$

end

Event *power_off* $\widehat{=}$

extends *power_off*

Event *power_on* $\widehat{=}$

extends *power_on*

Event *mount* $\widehat{=}$

extends *mount*

any

$objname$ the name of each object being mounted
$cdate$ the creation date of each object being mounted
$mdate$ the last modification date of each object being mounted
$fsize$ the size of each file being mounted

where

grd18 : $objname \in objs \rightarrow NAME$
grd19 : $cdate \in objs \rightarrow DATE$
grd20 : $mdate \in objs \rightarrow DATE$
grd21 : $fsize \in fs \rightarrow \mathbb{N}$
grd21 : $fsize = card(fat\_tmp(f))$

then
act8 : $oname := oname \cup objname$
act9 : $dateCreated := dateCreated \cup cdate$
act10 : $dateLastModified := dateLastModified \cup mdate$
act11 : $file\_size := file\_size \cup fsize$
end

Event $unmount \mathrel{\widehat{=}}$
extends $unmount$
then
act8 : $oname := objs \lhd oname$
act9 : $dateCreated := objs \lhd dateCreated$
act10 : $dateLastModified := objs \lhd dateLastModified$
act11 : $file\_size := objs \lhd file\_size$
end

END

A.6 The fifth refinement: Decomposition of the *writefile* event

Decomposing the *writefile* event into sub events: *w_start*, *w_step* and *w_end*.

MACHINE FMCH05
REFINES FMCH04
SEES FCTX03
VARIABLES

...
$writing$
$fcont\_tmp$

INVARIANTS

inv1 : $writing \subseteq w\_opened\_files$
inv2 : $fcont\_tmp \in writing \rightarrow CONTENT$
inv3 : $\forall f \cdot f \in writing \Rightarrow fcont\_tmp(f) \subseteq wbuffer(f)$
inv4 : $power\_on = FALSE \Rightarrow writing := \varnothing \wedge fcont\_tmp := \varnothing$

EVENTS
Initialisation

extended

begin
act20 : $writing := \varnothing$
act21 : $fcont\_tmp := \varnothing$
end

Event *mkdir* $\widehat{=}$

Make a directory

extends *mkdir*

Event *crt_file* $\widehat{=}$

create file

extends *crt_file*

Event *move* $\widehat{=}$

Move an object from one place to another.

extends *move*

Event *delfile* $\widehat{=}$

Delete an object, and all its descendents, from specified directory.

extends *delfile*

Event *rmdir* $\widehat{=}$

Delete an empty directory.

extends *rmdir*

Event *deltree* $\widehat{=}$

extends *deltree*

Event *copy* $\widehat{=}$

Copy an existing object and its descendants to another place

extends *copy*

Event *w_open* $\widehat{=}$

Open file for writing.

extends *w_open*

Event *r_open* $\widehat{=}$

Open file for reading

extends *r_open*

Event *readfile* $\widehat{=}$

Read the whole content of a file from the storage into the read buffer.

extends *readfile*

Event *w_start* $\widehat{=}$

Start writing file f.

any
f
where
grd1 : $f \in w\_opened\_files$
grd2 : $f \notin writing$
grd3 : $powerloss = FALSE$
then
act1 : $writing := writing \cup \{f\}$
act2 : $fcont\_tmp(f) := \varnothing$
end

Event *w_step* $\widehat{=}$

Writing step, write a *data* of page i from the buffer into *fcont_tmp* (which is a mirror of the storage)

any
$f, i, data$
where
grd1 : $f \in writing$
grd2 : $i \in \mathbb{N}$
grd3 : $data \in DATA$
grd4 : $i \mapsto data \in wbuffer(f)$
grd5 : $i \notin dom(fcont\_tmp(f))$
grd6 : $powerloss = FALSE$
then
act1 : $fcont\_tmp(f) := fcont\_tmp(f) \cup \{i \mapsto data\}$
end

Event *w_end_ok* $\widehat{=}$

Writing a file is completed when all pages have been written (*grd3*)

refines *writefile*

any
f
sz
where
grd1 : $f \in writing$
grd2 : $powerloss = FALSE$
grd3 : $dom(fcont\_tmp(f)) = dom(wbuffer(f))$

grd4 : $sz \in \mathbb{N}$

then

act1 : $fcontent(f) := fcont\_tmp(f)$
act2 : $dateLastModified(f) := nowdate$
act3 : $writing := writing \setminus \{f\}$
act4 : $fcont\_tmp := \{f\} \lhd fcont\_tmp$
act5 : $file\_size(f) := card(fcont\_tmp(f))$

end

Event *w_end_fail* $\widehat{=}$

Writing a file fails. Release all memory contents.

any

f

where

grd1 : $f \in writing$

then

act1 : $writing := writing \setminus \{f\}$
act2 : $fcont\_tmp := \{f\} \lhd fcont\_tmp$

end

Event *close* $\widehat{=}$

Close an opened file.

extends *close*

where

grd3 : $f \notin writing$

end

Event *rename* $\widehat{=}$

extends *rename*

Event *power_off* $\widehat{=}$

extends *power_off*

then

act6 : $writing := \varnothing$
act7 : $fcont\_tmp := \varnothing$

end

Event *power_on* $\widehat{=}$

extends *power_on*

Event *mount* $\widehat{=}$

extends *mount*

Event *unmount* $\widehat{=}$

extends *unmount*

END

A.7 The sixth refinement: Decomposition of the *readfile* event

MACHINE FMCH06
REFINES FMCH05
SEES FCTX03
VARIABLES

...
$reading$
$rbuff\_tmp$

INVARIANTS

inv1 : $reading \subseteq r\_opened\_files$
inv2 : $rbuff\_tmp \in reading \rightarrow CONTENT$
inv3 : $\forall f \cdot f \in reading \Rightarrow rbuff\_tmp(f) \subseteq fcontent(f)$
inv4 : $power\_on = FALSE \Rightarrow reading := \varnothing \land rbuff\_tmp := \varnothing$
thm1 : $\forall f \cdot f \in reading \Rightarrow f \in dom(fcontent)$

EVENTS
Initialisation

extended
begin
act22 : $reading := \varnothing$
act23 : $rbuff\_tmp := \varnothing$
end

Event *mkdir* $\widehat{=}$

Make a directory
extends *mkdir*

Event *crt_file* $\widehat{=}$

create file
extends *crt_file*

Event *move* $\widehat{=}$

Move an object from one place to another.
extends *move*

Event *delfile* $\widehat{=}$

Delete an object, and all its descendants, from specified directory.
extends *delfile*

Event *rmdir* $\widehat{=}$

Delete an object, and all its descendents, from specified directory.
extends *rmdir*

Event *deltree* $\widehat{=}$

extends *deltree*

Event *copy* $\widehat{=}$

Copy an existing object
extends *copy*

Event *w_open* $\widehat{=}$

Open file for writing.

extends *w_open*

Event *r_open* $\widehat{=}$

Open file for reading

extends *r_open*

Event *w_start* $\widehat{=}$

Start writing a file.

extends *w_start*

Event *w_step* $\widehat{=}$

Writing step, write one data unit from a buffer into fcont_tmp (which is a mirror of the storage)

extends *w_step*

Event *w_end_ok* $\widehat{=}$

extends *w_end_ok*

Event *w_end_fail* $\widehat{=}$

Writing of a file fails.

extends *w_end_fail*

Event *close* $\widehat{=}$

Close an opened file.

extends *close*

where

grd3 : $f \notin reading$

end

Event *rename* $\widehat{=}$

extends *rename*

Event *r_start* $\widehat{=}$

Start reading of the given file

any

f

where

grd1 : $f \in r\_opened\_files$

grd2 : $f \notin reading$

then

act1 : $reading := reading \cup \{f\}$

act2 : $rbuff\_tmp(f) := \varnothing$

end

Event *r_step* $\widehat{=}$

Reading step, read the data of page i from the storage into the temp buffer.

any

$f, i, data$

where

grd1 : $f \in reading$

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

grd3 : $data \in DATA$

grd4 : $i \mapsto data \in fcontent(f)$

grd5 : $i \notin dom(rbuff\_tmp(f))$

then

act1 : $rbuff\_tmp(f) := rbuff\_tmp(f) \cup \{i \mapsto data\}$

end

Event *r_end_ok* $\widehat{=}$

Reading the whole content of file f from the storage into the read buffer is completed when all pages have been read (*grd*2).

refines *readfile*

any

f

where

grd1 : $f \in reading$

grd2 : $dom(rbuff\_tmp(f)) = dom(fcontent(f))$

grd3 : $powerloss = FALSE$

then

act1 : $rbuffer(f) := rbuff\_tmp(f)$

act2 : $reading := reading \setminus \{f\}$

act3 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$

end

Event *r_end_fail* $\widehat{=}$

Reading of the given file fails. This event releases all memory buffers

any

f

where

grd1 : $f \in reading$

then

act1 : $reading := reading \setminus \{f\}$

act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$

end

Event *power_off* $\widehat{=}$

extends *power_off*

then

act8 : $reading := \varnothing$

act9 : $rbuff\_tmp := \varnothing$

end

Event *power_on* $\widehat{=}$

extends *power_on*

Event *mount* $\widehat{=}$

extends *mount*

Event *unmount* $\widehat{=}$

extends *unmount*

END

A.8 The seventh refinement: Flash specification

Relating to flash interfaces provided. *fcontent* and *fcont_tmp* are replaced by fat and *fat_tmp*. Note: Because copying can be done recursively by events read and write, we decided not to refine it in this level.

MACHINE FMCH07
REFINES FMCH06
SEES FLCTX
VARIABLES

`files`
`directories`
`parent`
`w_opened_files` A set of files being opened for writing
`r_opened_files` A set of files being opened for reading
`wbuffer` Write buffer of each w_opened files, containing the content to be written to the flash
`rbuffer` Read buffer of each file being opened of reading
`power_on`
`users`
`groups`
`user_grps`
`user_pgrp`
`obj_owner`
`obj_grp`
`obj_perms`
`oname`
`dateCreated`
`dateLastModified`
`file_size`
`writing` Set of files being in writing state
`reading` Set of files being in reading state
`rbuff_tmp` Temporary read buffer. It becomes the actual output buffer when all pages has been read into the memory.
fat The table of contents of each file.
fat_tmp Temporary fat.
$curr\_version$ The current version of each file
$writing\_version$ Writing version of each file
$flash$ A flash device which is an array of pages
$programmed\_pages$ Set of pages that have already been programmed
$obsolete\_pages$ Set of programmed pages which are obsolete.

INVARIANTS

inv1 : $flash \in RowAddr \rightarrow PDATA$
inv2 : $programmed\_pages \subseteq RowAddr$
inv3 : $obsolete\_pages \subseteq programmed\_pages$

inv4 : $power\_on = TRUE \Rightarrow fat \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr)$

This fat is a mapping of each file to a table that maps each page index within the file to its corresponding row address in the flash.

inv5 : $fat\_tmp \in writing \rightarrow (\mathbb{N} \nrightarrow RowAddr)$

inv6 : $curr\_version \in (files \cup directories) \rightarrow VERNUM$

inv7 : $writing\_version \in writing \rightarrow VERNUM$

inv8 : $\forall p \cdot p \in PDATA \wedge objOfpage(p) \in files$
$\wedge\ verOfpage(p) = curr\_version(objOfpage(p)) \wedge pidxOfpage(p) \neq 0$
$\Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcontent(objOfpage(p))$

inv9 : $\forall i, r, f, p \cdot f \in files \wedge r \in programmed\_pages \setminus obsolete\_pages$
$\wedge\ p = flash(r) \wedge verOfpage(p) = curr\_version(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0$
$\Rightarrow\ \ i \mapsto r \in fat(f)$

inv10 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages$
$\wedge\ f \in writing \wedge p = flash(r) \wedge verOfpage(p) = writing\_version(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0$
$\Rightarrow\ \ i \mapsto r \in fat\_tmp(f)$

inv11 : $\forall f \cdot f \in files \Rightarrow dom(fat(f)) = dom(fcontent(f))$

inv12 : $\forall f \cdot f \in files \Rightarrow dom(fcontent(f)) = 1 \,..\, file\_size(f)$

inv13 : $\forall f \cdot f \in writing \Rightarrow dom(fat\_tmp(f)) = dom(fcont\_tmp(f))$

inv14 : $power\_on = FALSE \Rightarrow fat := \varnothing \wedge fat\_tmp := \varnothing \wedge writing\_version := \varnothing$

inv15 : $\forall f \cdot f \in writing \Rightarrow writing\_version(f) \neq curr\_version(f)$

EVENTS

Initialisation

begin

act1 : $files := \varnothing$

act2 : $directories := \{root\}$

act3 : $parent := \varnothing$

act4 : $w\_opened\_files := \varnothing$

act5 : $r\_opened\_files := \varnothing$

act6 : $wbuffer := \varnothing$

act7 : $rbuffer := \varnothing$

act8 : $users := \{su\}$

act9 : $groups := \{admin\}$

act10 : $user\_grps := \{su \mapsto admin\}$

act11 : $user\_pgrp := \{su \mapsto admin\}$

act12 : $obj\_owner := \{root \mapsto su\}$

act13 : $obj\_grp := \{root \mapsto admin\}$

act14 : $obj\_perms := \{root \mapsto wbo, root \mapsto rbo, root \mapsto xbo\}$

act15 : $oname := \{root \mapsto rname\}$

act16 : $dateCreated := \{root \mapsto dfdate\}$

act17 : $dateLastModified := \{root \mapsto dfdate\}$

act18 : $writing := \varnothing$

act19 : $reading := \varnothing$

act20 : $rbuff\_tmp := \varnothing$

act21 : $fat := \varnothing$

act22 : $fat\_tmp := \varnothing$

act23 : $curr\_version := \varnothing$

act24 : $writing\_version := \varnothing$

act25 : $programmed\_pages := \varnothing$
act26 : $obsolete\_pages := \varnothing$
act27 : $flash := dflash$
act28 : $power\_on := TRUE$
act29 : $file\_size := \varnothing$

end

Event *mkdir* $\widehat{=}$

Make a directory

extends *mkdir*

any

r a row address used to record the new description of the object being created
$desc$ the description to be stored
$pdata$ a page data to be programmed at row r

where

grd10 : $r \in RowAddr \setminus programmed\_pages$
grd11 : $pdata \in PDATA$
grd12 : $desc \in DATA$
grd13 : $objOfpage(pdata) = obj$
grd14 : $pidxOfpage(pdata) = 0$
grd15 : $verOfpage(pdata) = 0$
grd16 : $dataOfpage(pdata) = desc$

then

act10 : $flash(r) := pdata$
act11 : $programmed\_pages := programmed\_pages \cup \{r\}$
act12 : $curr\_version(obj) := 0$

end

Event *crt_file* $\widehat{=}$

create file

refines *crt_file*

any

$obj, indr, usr, grp, nme$
r a row address used to record the new description of the file being created
$fdesc$ the description of file to be stored
$pdata$ a page data to be programmed at row r

where

grd1 : $obj \in OBJECT \setminus (files \cup directories)$
grd2 : $indr \in directories$
grd3 : $usr \in users$
grd4 : $grp \in groups$
grd5 : $usr \mapsto grp \in user\_pgrp$
grd6 : $indr \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
grd7 : $nme \in NAME$
grd8 : $nme \notin oname[parent^{-1}[\{indr\}]]$
grd9 : $fdesc \in DATA$
grd10 : $pdata \in PDATA$
grd11 : $r \in RowAddr \setminus programmed\_pages$
grd12 : $objOfpage(pdata) = obj$
grd13 : $pidxOfpage(pdata) = 0$
grd14 : $verOfpage(pdata) = 0$
grd15 : $dataOfpage(pdata) = fdesc$
grd16 : $\forall p \cdot p \in PDATA \land objOfpage(p) = obj \Rightarrow pidxOfpage(p) = 0$

grd17 : $power\_on = TRUE$

then

act1 : $files := files \cup \{obj\}$
act2 : $parent(obj) := indr$
act3 : $fat(obj) := \varnothing$
act4 : $obj\_owner(obj) := usr$
act5 : $obj\_grp(obj) := grp$
act6 : $obj\_perms := obj\_perms \cup \{obj \mapsto rbo, obj \mapsto wbo, obj \mapsto xbo\}$
act7 : $oname(obj) := nme$
act8 : $dateCreated(obj) := nowdate$
act9 : $dateLastModified(obj) := nowdate$
act10 : $curr\_version(obj) := 0$
act11 : $flash(r) := pdata$
act12 : $programmed\_pages := programmed\_pages \cup \{r\}$
act13 : $file\_size(obj) := 0$

end

Event *move* $\widehat{=}$

Move an object *obj* and its descendants *des* to another location *to* by user *usr*.

extends *move*

any

r the selected row address within the flash device to be written
$fdesc$ represents a DATA of file description (name, owner, permissions, etc.)
$pdata$ a PDATA to be written to flash

where

grd9 : $r \in RowAddr \setminus programmed\_pages$
grd10 : $pdata \in PDATA$
grd11 : $desc \in DATA$
grd12 : $objOfpage(pdata) = obj$
grd13 : $pidxOfpage(pdata) = 0$
grd14 : $verOfpage(pdata) = curr\_version(obj)$
grd15 : $dataOfpage(pdata) = desc$

then

act2 : $flash(r) := pdata$
act3 : $programmed\_pages := programmed\_pages \cup \{r\}$

end

Event *delfile* $\widehat{=}$

Delete a file.

refines *delfile*

any

obj
usr
$rows$ all row addresses belonging to object being deleted

where

grd1 : $obj \in files \setminus \{root\}$
grd2 : $parent^{-1}[\{obj\}] = \varnothing$
grd3 : $obj \notin w\_opened\_files \cup r\_opened\_files$
grd4 : $usr \in users$
grd5 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
grd6 : $obj \notin reading$
grd7 : $rows \subseteq programmed\_pages \setminus obsolete\_pages$
grd8 : $rows = flash^{-1}[objOfpage^{-1}[\{obj\}]]$
grd9 : $power\_on = TRUE$

then

act1 : $files := files \setminus \{obj\}$
act2 : $parent := \{obj\} \lhd parent$
act3 : $fat := \{obj\} \lhd fat$
act4 : $obj\_owner := \{obj\} \lhd obj\_owner$
act5 : $obj\_grp := \{obj\} \lhd obj\_grp$
act6 : $obj\_perms := \{obj\} \lhd obj\_perms$
act7 : $oname := \{obj\} \lhd oname$
act8 : $dateCreated := \{obj\} \lhd dateCreated$
act9 : $dateLastModified := \{obj\} \lhd dateLastModified$
act10 : $fat\_tmp := \{obj\} \lhd fat\_tmp$
act11 : $curr\_version := \{obj\} \lhd curr\_version$
act12 : $obsolete\_pages := obsolete\_pages \lhd rows$
act15 : $file\_size := \{obj\} \lhd file\_size$

end

Event *rmdir* $\widehat{=}$

Delete an empty directory.

extends *rmdir*

any
$rows$ all rows belonging to the *obj* being deleted
where
grd6 : $rows \subseteq programmed\_pages \setminus obsolete\_pages$
grd7 : $rows = flash^{-1}[objOfpage^{-1}[\{obj\}]]$
then
act9 : $obsolete\_pages := obsolete\_pages \cup rows$
end

Event *deltree* $\widehat{=}$

Delete the given object (*obj*) and all its descendants (*des*), by user *usr*.

refines *deltree*

any
$obj, des, objs, usr$
$rows$ all rows belonging to those objects (*objs*) being deleted.
where
grd1 : $obj \in (files \cup directories) \setminus \{root\}$
grd2 : $des \subseteq (files \cup directories)$
grd3 : $des = (tcl(parent))^{-1}[\{obj\}]$
grd4 : $objs = des \cup \{obj\}$
grd5 : $objs \cap (w\_opened\_files \cup r\_opened\_files) = \varnothing$
All must not be in use.
grd6 : $usr \in users$
grd7 : $obj \mapsto usr \in WPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
grd8 : $rows \subseteq programmed\_pages \setminus obsolete\_pages$
grd9 : $rows = flash^{-1}[objOfpage^{-1}[objs]]$
grd10 : $power\_on = TRUE$
then
act1 : $parent := objs \lhd parent$
act2 : $files := files \setminus (objs \cap files)$
act3 : $directories := directories \setminus (objs \cap directories)$
act4 : $fat := objs \lhd fat$
act5 : $obj\_owner := objs \lhd obj\_owner$
act6 : $obj\_grp := objs \lhd obj\_grp$
act7 : $obj\_perms := objs \lhd obj\_perms$
act8 : $oname := objs \lhd oname$
act9 : $dateCreated := objs \lhd dateCreated$

act10 : $dateLastModified := objs \lhd dateLastModified$
act11 : $obsolete\_pages := obsolete\_pages \cup rows$
act12 : $curr\_version := objs \lhd curr\_version$
act13 : $file\_size := objs \lhd file\_size$

end

Event $w\_open \mathrel{\widehat{=}}$

Open a file of writing.

extends w_open

Event $r\_open \mathrel{\widehat{=}}$

Open a file for reading

extends r_open

Event $w\_start \mathrel{\widehat{=}}$

Start writing of a file.

refines w_start

any

f
cv the current version of file f
wv the version of file f being written

where

grd1 : $f \in w\_opened\_files$
grd2 : $f \notin writing$
grd3 : $cv = curr\_version(f)$
grd4 : $wv \in VERNUM$
grd5 : $((wv = cv + 1 \Leftrightarrow cv < 2) \lor (wv = 0))$
grd6 : $\forall p \cdot p \in PDATA \land objOfpage(p) = f \Rightarrow verOfpage(p) \neq wv$
grd7 : $power\_on = TRUE$

then

act1 : $writing := writing \cup \{f\}$
act2 : $fat\_tmp(f) := \varnothing$
act3 : $writing\_version(f) := wv$

end

Event $w\_step \mathrel{\widehat{=}}$

Writing step, write the data of page i from the buffer into the flash device at row r.

refines w_step

any

f, i
$data$ the data of page i of file f
r the address to be written
$pdata$ the page data to be written to the flash

where

grd1 : $f \in writing$
grd2 : $i \in \mathbb{N}$
grd3 : $i > 0$
grd4 : $data \in DATA$
grd5 : $i \mapsto data \in wbuffer(f)$
grd6 : $i \notin dom(fat\_tmp(f))$
grd7 : $r \in RowAddr \setminus programmed\_pages$
grd8 : $pdata \in PDATA$
grd9 : $verOfpage(pdata) = writing\_version(f)$
grd10 : $pidxOfpage(pdata) = i$

grd11 : $objOfpage(pdata) = f$
grd12 : $dataOfpage(pdata) = data$
grd13 : $power\_on = TRUE$

then

act1 : $fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
act2 : $flash(r) := pdata$
act3 : $programmed\_pages := programmed\_pages \cup \{r\}$

end

Event $w\_end\_ok \mathrel{\widehat{=}}$

Writing the given file is complete when all pages have been written to the flash device.

refines w_end_ok

any

f
wv writing version
$data$ Contains file description
r row address to store file description
$pdata$ a PDATA to be programmed to row r

where

grd1 : $f \in writing$
grd2 : $dom(fat\_tmp(f)) = dom(wbuffer(f))$
grd3 : $wv = writing\_version(f)$
grd4 : $r \in RowAddr \setminus programmed\_pages$
grd5 : $data \in DATA$
grd6 : $pdata \in PDATA$
grd7 : $verOfpage(pdata) = wv$
grd8 : $pidxOfpage(pdata) = 0$
grd9 : $objOfpage(pdata) = f$
grd10 : $dataOfpage(pdata) = data$
grd11 : $power\_on = TRUE$

then

act1 : $fat(f) := fat\_tmp(f)$
act2 : $dateLastModified(f) := nowdate$
act3 : $writing := writing \setminus \{f\}$
act4 : $fat\_tmp := \{f\} \lhd fat\_tmp$
act5 : $curr\_version(f) := wv$
act6 : $writing\_version := \{f\} \lhd writing\_version$
act7 : $flash(r) := pdata$
act8 : $programmed\_pages := programmed\_pages \cup \{r\}$
act9 : $file\_size(f) := card(fat\_tmp(f))$

end

Event $w\_end\_fail \mathrel{\widehat{=}}$

Writing of a file fails

refines w_end_fail

any

f

where

grd1 : $f \in writing$

then

act1 : $writing := writing \setminus \{f\}$
act2 : $fat\_tmp := \{f\} \lhd fat\_tmp$
act3 : $writing\_version := \{f\} \lhd writing\_version$

end

Event *close* $\widehat{=}$

Close an opened file.

extends *close*

Event *rename* $\widehat{=}$

Rename the given object.

extends *rename*

any

$r, newdesc, pdata$

where

grd7 : $r \in RowAddr \setminus programmed\_pages$
grd8 : $pdata \in PDATA$
grd9 : $newdesc \in DATA$
grd10 : $objOfpage(pdata) = obj$
grd11 : $pidxOfpage(pdata) = 0$
grd12 : $verOfpage(pdata) = curr\_version(obj)$
grd13 : $dataOfpage(pdata) = newdesc$

then

act3 : $flash(r) := pdata$
act4 : $programmed\_pages := programmed\_pages \cup \{r\}$

end

Event *r_start* $\widehat{=}$

Start read the given file

refines *r_start*

any

f

where

grd1 : $f \in r\_opened\_files$
grd2 : $f \notin reading$
grd3 : $power\_on = TRUE$

then

act1 : $reading := reading \cup \{f\}$
act2 : $rbuff\_tmp(f) := \varnothing$

end

Event *r_step* $\widehat{=}$

Reading step, read the data of page i from the flash at row r into the temp buffer.

refines *r_step*

any

$f, i, data, r, pdata$

where

grd1 : $f \in reading$
grd2 : $i \in \mathbb{N}$
grd3 : $i > 0$
grd4 : $data \in DATA$
grd5 : $power\_on = TRUE$
grd6 : $r \in programmed\_pages \setminus obsolete\_pages$
grd7 : $i \mapsto r \in fat(f)$
grd8 : $pdata = flash(r)$
grd9 : $verOfpage(pdata) = curr\_version(f)$
grd10 : $pidxOfpage(pdata) = i$
grd11 : $objOfpage(pdata) = f$
grd12 : $data = dataOfpage(pdata)$

grd13 : $i \notin dom(rbuff\_tmp(f))$
grd14 : $i \mapsto data \notin rbuff\_tmp(f)$

then

act1 : $rbuff\_tmp(f) := rbuff\_tmp(f) \cup \{i \mapsto data\}$

end

Event *r_end_ok* $\widehat{=}$

Reading the given file is completed when all pages have been read (*grd*3).

refines *r_end_ok*

any

f

where

grd1 : $power\_on = TRUE$
grd2 : $f \in reading$
grd3 : $dom(rbuff\_tmp(f)) = dom(fat(f))$

then

act1 : $rbuffer(f) := rbuff\_tmp(f)$
act2 : $reading := reading \setminus \{f\}$
act3 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$

end

Event *r_end_fail* $\widehat{=}$

Reading of a file fails.

refines *r_end_fail*

any

f

where

grd1 : $f \in reading$

then

act1 : $reading := reading \setminus \{f\}$
act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$

end

Event *mark_obsolete* $\widehat{=}$

A utility event to mark all pages (identified by *rows*) of the given file that has the version number equal to the given version (*ver*).

any

$f, ver, rows$

where

grd1 : $f \in files$
grd2 : $ver \in VERNUM$
grd3 : $rows \subseteq programmed\_pages$
grd4 : $rows = flash^{-1}[objOfpage^{-1}[\{f\}] \cap verOfpage^{-1}[\{ver\}]]$
grd5 : $power_on = TRUE$

then

act1 : $obsolete\_pages := obsolete\_pages \cup rows$

end

Event *power_off* $\widehat{=}$

refines *power_off*

when

grd1 : $power\_on = TRUE$

then

act1 : $power\_on := FALSE$
act2 : $wbuffer := \varnothing$
act3 : $rbuffer := \varnothing$
act4 : $w\_opened\_files := \varnothing$

act5 : $r\_opened\_files := \varnothing$
act6 : $fat := \varnothing$
act7 : $writing := \varnothing$
act8 : $fat\_tmp := \varnothing$
act9 : $reading := \varnothing$
act10 : $rbuff\_tmp := \varnothing$
act11 : $writing\_version := \varnothing$

end

Event *power_on* $\widehat{=}$

Reconstructs a FAT table.

refines *power_on*

any
ft

where
grd1 : power_on = FALSE
grd2 : $ft \in files \to (\mathbb{N} \nrightarrow RowAddr)$
grd3 : $\forall f \cdot f \in files \Rightarrow dom(ft(f)) = 1 \,..\, file\_size(f)$
grd4 : $\forall p \cdot p \in PDATA \wedge objOfpage(p) \in dom(ft) \Rightarrow p \in ran(flash)$
grd5 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in files$
$\wedge\ p = flash(r) \wedge verOfpage(p) = curr\_version(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i \wedge i \neq 0$
$\Rightarrow\ i \mapsto r \in ft(f)$
grd6 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in files$
$\wedge\ p = flash(r) \wedge i \mapsto r \in ft(f)$
$\Rightarrow\ (\ verOfpage(p) = curr\_version(f) \wedge$
$objOfpage(p) = f \wedge pidxOfpage(p) = i\ )$

then
act1 : power_on := TRUE
act2 : $fat := ft$

end

Event *mount* $\widehat{=}$

Mount the flash contents into an existing root.

refines *mount*

any
objs set of all objects to be mounted.
fs Set of files to be mounted.
ds set of directories to be mounted.
prt parent function representing the structure of all *objs* to be mounted.
x Subroot to be mounted to the existing root.
$fcnt$ the content of each file
objown the owner of each object being mounted
objperms the set of permissions of each object being mounted
objgrp the group owner of each object being mounted
objname the name of each object being mounted
$cdate$ the creation date of each object being mounted
$mdate$ the modification date of each object being mounted
$fsize$ the size of each file being mounted
ft the table of content of each file being mounted
cv the current version of each object being mounted

where
grd1 : $objs \subseteq OBJECT$
grd2 : $fs \subseteq objs$
grd3 : $ds \subseteq objs$
grd4 : $objs = fs \cup ds$

grd5 : $fs \cap ds = \varnothing$
grd6 : $(files \cup directories) \cap objs = \varnothing$
grd7 : $x \in ds$
grd8 : $prt \in objs \setminus \{x\} \rightarrow ds$
grd9 : $\forall s \cdot (s \subseteq prt^{-1}[s] \Rightarrow s = \varnothing)$
grd10 : $prt \cap parent = \varnothing$
grd11 : $files \cap fs = \varnothing$
grd12 : $directories \cap ds = \varnothing$
grd13 : $fcnt \in fs \rightarrow CONTENT$
grd14 : $objown \in objs \rightarrow users$
grd15 : $objperms \in objs \leftrightarrow PERMISSION$
grd16 : $objgrp \in objs \rightarrow groups$
grd17 : $objname \in objs \rightarrow NAME$
grd18 : $cdate \in objs \rightarrow DATE$
grd19 : $mdate \in objs \rightarrow DATE$
grd20 : $fsize \in fs \rightarrow \mathbb{N}$
grd21 : $ft \in fs \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
grd22 : $cv \in objs \rightarrow VERNUM$
grd23 : $\forall p \cdot p \in ran(flash) \wedge objOfpage(p) \in fs \wedge verOfpage(p) = cv(objOfpage(p))$
$\wedge\ pidxOfpage(p) \neq 0 \Rightarrow pidxOfpage(p) \mapsto dataOfpage(p) \in fcnt(objOfpage(p))$
grd24 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in fs$
$\wedge\ p = flash(r) \wedge verOfpage(p) = curr\_version(f)$
$\wedge\ objOfpage(p) = f \wedge pidxOfpage(p) = i$
$\Rightarrow\ i \mapsto r \in ft(f)$
grd25 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \wedge f \in fs$
$\wedge\ p = flash(r) \wedge i \mapsto r \in ft(f)$
$\Rightarrow\ (\ verOfpage(p) = cv(f) \wedge$
$objOfpage(p) = f \wedge pidxOfpage(p) = i\ )$
grd26 : $\forall f \cdot f \in fs \Rightarrow dom(ft(f)) = dom(fcnt(f))$
grd27 : $power\_on = TRUE$

then

act1 : $files := files \cup fs$
act2 : $directories := directories \cup ds$
act3 : $parent := parent \cup prt \cup \{x \mapsto root\}$
act4 : $fat := fat \cup ft$
act5 : $obj\_owner := obj\_owner \cup objown$
act6 : $obj\_perms := obj\_perms \cup objperms$
act7 : $obj\_grp := obj\_grp \cup objgrp$
act8 : $oname := oname \cup objname$
act9 : $dateCreated := dateCreated \cup cdate$
act10 : $dateLastModified := dateLastModified \cup mdate$
act11 : $file\_size := file\_size \cup fsize$
act12 : $curr\_version := curr\_version \cup cv$

end

Event *unmount* $\widehat{=}$

unmount a flash device consisting of objects *objs* rooted at x

refines *unmount*

any

$objs, x$

where

grd1 : $objs \subseteq files \cup directories$
grd2 : $root \notin objs$
grd3 : $x \in objs$

grd5 : $objs = (tcl(parent))^{-1}[\{x\}] \cup \{x\}$
grd6 : $objs \cap w\_opened\_files = \varnothing$
no files are in used
grd7 : $objs \cap r\_opened\_files = \varnothing$
no files are in used
grd8 : $power\_on = TRUE$

then

act1 : $files := files \setminus (objs \cap files)$
act3 : $directories := directories \setminus (objs \cap directories)$
act2 : $parent := objs \lhd parent$
act4 : $fat := objs \lhd fat$
act5 : $obj\_owner := objs \lhd obj\_owner$
act6 : $obj\_grp := objs \lhd obj\_grp$
act7 : $obj\_perms := objs \lhd obj\_perms$
act8 : $oname := objs \lhd oname$
act9 : $dateCreated := objs \lhd dateCreated$
act10 : $dateLastModified := objs \lhd dateLastModified$
act11 : $file\_size := objs \lhd file\_size$
act12 : $curr\_version := objs \lhd curr\_version$

end

END

A.9 Contexts

CONTEXT FCTX

Defines objects, root object and transitive closure of relations on objects, and introduces some theorems used for discharging OPs.

SETS

`OBJECT`

CONSTANTS

`root` root object
`objrel` type of relation on objects
`tcl` transitive closure of an objrel
`objfn` type of function on objects

AXIOMS

axm1 : $root \in OBJECT$
axm2 : $objfn = OBJECT \setminus \{root\} \pfun OBJECT$
axm4 : $objrel = OBJECT \rel OBJECT$
axm3 : $tcl \in objrel \fun objrel$
axm5 : $\forall r \cdot (r \in objrel \limp r \subseteq tcl(r))$
r included in tcl(r)
axm6 : $\forall r \cdot (r \in objrel \limp r;\ tcl(r) \subseteq tcl(r))$
unfolding included in tcl(r)
axm7 : $\forall r, t \cdot (r \in objrel \land r \subseteq t \land r;\ t \subseteq t \limp tcl(r) \subseteq t)$
tcl(r) is least
thm5 : $objfn \subseteq objrel$
thm1 : $\forall r \cdot r \in objrel \limp tcl(r) = r \cup (r;\ tcl(r))$
tcl(r) is a fixed point
thm2 : $\forall t \cdot t \in objfn \land (\forall s \cdot s \subseteq t^{-1}[s] \limp s = \varnothing) \limp tcl(t) \cap (OBJECT \domres id) = \varnothing$
No loop theorem: $(!s.s <: (t^{-1})[s] => s = \{\})$ imples tcl(t) has no loops.
thm3 : $tcl(\varnothing) = \varnothing$

END

CONTEXT FCTX01

including additional theorems used for discharging POs

EXTENDS FCTX

AXIOMS

thm1 : $\forall r, r2 \cdot$

$r \in objrel$
$\land\ r2 \in objrel$
$\land\ r2 \subseteq r$
$\land\ (\forall s \cdot s \subseteq r[s] \Rightarrow s = \varnothing)$
$\Rightarrow$
$(\forall t \cdot t \subseteq (r2)[t] \Rightarrow t = \varnothing)$

This thm is used for delete operation, when r2 is a tree after delete. This thm is used to prove inv10 (no-loop property).

thm2 : $\forall f, g, c, t, u, M, N \cdot$

$N \subseteq OBJECT$
$\land\ M \subseteq OBJECT$
$\land\ N \cap M = \varnothing$
$\land\ t \in M$
$\land\ f \in M \setminus \{t\} \to M$
$\land\ u \in N$
$\land\ c \in M \rightarrowtail\!\!\!\!\!\rightarrow N$
$\land\ u = c(t)$
$\land\ g = (c^{-1};\ f;\ c)$
$\Rightarrow$
$g \in N \setminus \{u\} \to N$

This is used to prove that g is a total function (g is a copy of a subtree f rooted at t; and c is a corresponding function). $M = objs$ (all objects being copied). $N = nobjs$ (new objects which are copies). $t =$ a root node of subtree f. $f = des \lhd parent$ (des is a set of all descendants from t). $c = objs \rightarrowtail\!\!\!\!\!\rightarrow nobjs$. $u =$ the correspondent of t. This is used for event copy, move and create.

thm3 : $\forall f, c, g, t, u, M, N \cdot$

$N \subseteq OBJECT$
$\land\ M \subseteq OBJECT$
$\land\ t \in M$
$\land\ f \in M \setminus \{t\} \to M$
$\land\ (\forall s \cdot s \subseteq f^{-1}[s] \Rightarrow s = \varnothing)$
$\land\ u \in N$
$\land\ c \in M \rightarrowtail\!\!\!\!\!\rightarrow N$
$\land\ u = c(t)$
$\land\ g = c^{-1};\ f;\ c$
$\land\ g \in N \setminus \{u\} \to N$
$\Rightarrow (\forall w \cdot w \subseteq g^{-1}[w] \Rightarrow w = \varnothing)$

This thm is used to prove that there is no-loop in g. It is used for copy, create, move operations.

thm4 : $\forall f, g, t, u, x, M, N \cdot$

$$
\begin{aligned}
& N \subseteq OBJECT \\
& \land M \subseteq OBJECT \\
& \land N \cap M = \varnothing \\
& \land t \in M \\
& \land f \in M \setminus \{t\} \to M \\
& \land u \in N \\
& \land g \in N \setminus \{u\} \to N \\
& \land x \in M \\
& \Rightarrow \\
& f \cup g \cup \{u \mapsto x\} \in (M \cup N) \setminus \{t\} \to M \cup N
\end{aligned}
$$

This thm is used to prove inv8, and is used by thm14 for copying (also for inserting and moving) By providing f = an original tree (= *parent* for copying and inserting, = $(des \cup \{obj\}) \lhd parent$ for moving) M = set of all objects in f. (= *objects* for copying and inserting, = $objects \setminus (des \cup \{obj\})$ for moving) N = set of new objects being added. (= *nobjs* for copying, = $\{obj\}$ for inserting, = $des \cup \{obj\}$ for moving) g = a copy of a subtree of f. (= $corres^{-1}$; $des \lhd parent$; *corres* for copying, = $\{\}$ for inserting, = $des \lhd parent$ for moving). t = *root* node. u = an object being copied, inserted or moved (obj). x = a target location (or *parent*).

thm5 : $\forall f, g, t, u, x, M, N \cdot$

$$
\begin{aligned}
& N \subseteq OBJECT \\
& \land M \subseteq OBJECT \\
& \land N \cap M = \varnothing \\
& \land t \in M \\
& \land f \in M \setminus \{t\} \to M \\
& \land u \in N \\
& \land g \in N \setminus \{u\} \to N \\
& \land x \in M \\
& \land (\forall A \cdot A \subseteq f^{-1}[A] \Rightarrow A = \varnothing) \\
& \land (\forall B \cdot B \subseteq g^{-1}[B] \Rightarrow B = \varnothing) \\
& \land f \cup g \cup \{u \mapsto x\} \in (M \cup N) \setminus \{t\} \to M \cup N \\
& \Rightarrow \\
& (\forall C \cdot C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \Rightarrow C = \varnothing)
\end{aligned}
$$

This thm is used for copying, inserting and moving in order to maintain inv10. For copying, we give: $f = parent, g = des <| parent, u = obj, x = to, M = objects, N = nobjs$ This theorem can be used for inserting a new object by providing: $f = parent, g = \{\}, u = obj, x = indr, M = objects, N = \{obj\}$, (u is an object being inserted into the location x). We can use this theorem for moving by providing: $f = des\{obj\} <<| parent, g = des <| parent, t =$

$rootu = obj, x = to, M = objects\ (des\{obj\})N = des\{obj\}$ (u is an object being moved to the location x)

END

CONTEXT FCTX02

EXTENDS FCTX01

SETS

DATA
NAME
DATE

CONSTANTS

CONTENT
rname
dfdate
nowdate
sizeOfdata

AXIOMS

axm1 : $CONTENT = \mathbb{N} \pfun DATA$
axm2 : $\varnothing \in CONTENT$
axm3 : $rname \in NAME$
axm4 : $dfdate \in DATE$
axm5 : $nowdate \in DATE$
axm6 : $sizeOfdata \in DATA \tfun \mathbb{N}$
axm7 : $\forall c \cdot c \in CONTENT \limp finite(c)$

END

CONTEXT FCTX03

Introducing PERMISSION, USER and GROUP

EXTENDS FCTX02

SETS

`USER`
`GROUP`
`PERMISSION`

CONSTANTS

`admin` admin group
`su` super user
`rbo` read by owner
`wbo` written by owner
`xbo` executed by owner
`rbg` read by group
`wbg` written by group
`xbg` executed by group
`rbw` read by world
`wbw` written by world
`xbw` executed by world
`RPerm`
`WPerm`
`XPerm`

AXIOMS

axm1 : $PERMISSION = \{rbo, wbo, xbo, rbg, wbg, xbg, rbw, wbw, xbw\}$

axm2 : $WPerm \in (OBJECT \leftrightarrow PERMISSION) \times (OBJECT \nrightarrow USER) \times (OBJECT \nrightarrow GROUP) \times (USER \leftrightarrow GROUP) \rightarrow (OBJECT \leftrightarrow USER)$

axm3 : $RPerm \in (OBJECT \leftrightarrow PERMISSION) \times (OBJECT \nrightarrow USER) \times (OBJECT \nrightarrow GROUP) \times (USER \leftrightarrow GROUP) \rightarrow (OBJECT \leftrightarrow USER)$

axm4 : $XPerm \in (OBJECT \leftrightarrow PERMISSION) \times (OBJECT \nrightarrow USER) \times (OBJECT \nrightarrow GROUP) \times (USER \leftrightarrow GROUP) \rightarrow (OBJECT \leftrightarrow USER)$

axm5 : $\forall o, u, p, s, g, m \cdot o \in OBJECT \land u \in USER$
$\land\ p \in (OBJECT \leftrightarrow PERMISSION)$
$\land\ s \in (OBJECT \nrightarrow USER)$
$\land\ g \in (OBJECT \nrightarrow GROUP)$
$\land\ m \in (USER \leftrightarrow GROUP)$
$\land\ o \in dom(g)$
$\Rightarrow$
$($
$o \mapsto u \in WPerm(p \mapsto s \mapsto g \mapsto m)$
$\Leftrightarrow$
$($

$$
\begin{array}{l}
\quad (o \mapsto u \in s \land o \mapsto wbo \in p) \lor \\
\quad (g(o) \in m[\{u\}] \land o \mapsto wbg \in p) \lor \\
\quad (o \mapsto wbw \in p) \lor \\
\quad (u = su) \\
\quad) \\
) \\
\end{array}
$$

o is an object, u is a user, p: object-permission relation, s: object-owner function, g: object-group function, m: user-group relation

axm6 : $\forall o, u, p, s, g, m \cdot o \in OBJECT \land u \in USER$

$$
\begin{array}{l}
\land\ p \in (OBJECT \leftrightarrow PERMISSION) \\
\land\ s \in (OBJECT \nrightarrow USER) \\
\land\ g \in (OBJECT \nrightarrow GROUP) \\
\land\ m \in (USER \leftrightarrow GROUP) \\
\land\ o \in dom(g) \\
\Rightarrow \\
(\\
\quad o \mapsto u \in RPerm(p \mapsto s \mapsto g \mapsto m) \\
\quad \Leftrightarrow \\
\quad (\\
\quad (o \mapsto u \in s \land o \mapsto rbo \in p) \lor \\
\quad (g(o) \in m[\{u\}] \land o \mapsto rbg \in p) \lor \\
\quad (o \mapsto rbw \in p) \lor \\
\quad (u = su) \\
\quad) \\
) \\
\end{array}
$$

axm7 : $\forall o, u, p, s, g, m \cdot o \in OBJECT \land u \in USER$

$$
\begin{array}{l}
\land\ p \in (OBJECT \leftrightarrow PERMISSION) \\
\land\ s \in (OBJECT \nrightarrow USER) \\
\land\ g \in (OBJECT \nrightarrow GROUP) \\
\land\ m \in (USER \leftrightarrow GROUP) \\
\land\ o \in dom(g) \\
\Rightarrow \\
(\\
\quad o \mapsto u \in XPerm(p \mapsto s \mapsto g \mapsto m) \\
\quad \Leftrightarrow \\
\quad (\\
\quad (o \mapsto u \in s \land o \mapsto xbo \in p) \lor \\
\quad (g(o) \in m[\{u\}] \land o \mapsto xbg \in p) \lor \\
\quad (o \mapsto xbw \in p) \lor \\
\quad (u = su) \\
\quad) \\
\end{array}
$$

$$)$$

axm8 : $su \in USER$

axm9 : $admin \in GROUP$

END

CONTEXT FLCTX

EXTENDS FCTX03

SETS

`BYTE` Data item
`PDATA`
`RowAddr`

CONSTANTS

`FLASH` A collection of LUNs
`dflash` default target
`dp` default page data
`objOfpage`
`verOfpage`
`pidxOfpage`
`dataOfpage`
`VERNUM`

AXIOMS

`axm15` : $FLASH = RowAddr \to PDATA$
`axm18` : $dp \in PDATA$
`axm16` : $dflash \in FLASH$
`axm34` : $VERNUM = 0 \mathinner{..} 2$
`axm30` : $objOfpage \in PDATA \to OBJECT$
`axm31` : $verOfpage \in PDATA \to VERNUM$
`axm32` : $pidxOfpage \in PDATA \to \mathbb{N}$
`axm33` : $dataOfpage \in PDATA \to DATA$

END

Appendix B

An Event-B specification of a file system, V2

The specification given here is the revised version of the specification given in Appendix A. The revision is based on the requirement that have been changed (i.e. satisfying unbounded version number and partial write/read operations). Details have already discussed in Chapter 5. Because of the similarity between the original version and the revised version. We will give only part of the specification that have been affected (in the second, fifth, sixth and seventh refinements).

B.1 The second refinement: File content

There are two main parts that have been affected because of the changes of system requirements. Namely, events *readfile* and *writefile* that are required to support partial reading/writing files.

MACHINE FMCH02B
REFINES FMCH01
SEES FCTX02
VARIABLES

`files`
`directories`
`parent`
$fcontent$ the content of each file
w_opened_files files which are opened for writing
r_opened_files files which are opened for reading
$wbuffer$ writing buffers of w_opened_files
$rbuffer$ reading buffers of r_opened_files
$power\_on$ power status

INVARIANTS

inv1 : $power\_on \in BOOL$
inv2 : $power\_on = TRUE \Rightarrow fcontent \in files \rightarrow CONTENT$
inv3 : $w\_opened\_files \subseteq files$
inv4 : $r\_opened\_files \subseteq files$
inv5 : $w\_opened\_files \cap r\_opened\_files = \varnothing$
inv6 : $wbuffer \in w\_opened\_files \rightarrow CONTENT$
inv7 : $rbuffer \in r\_opened\_files \rightarrow CONTENT$
inv8 : $power\_on = FALSE \Rightarrow (w\_opened\_files = \varnothing \land r\_opened\_files = \varnothing$
$wbuffer = \varnothing \land rbuffer = \varnothing)$

EVENTS

Initialisation

extended

begin
act5 : $fcontent := \varnothing$
act6 : $w\_opened\_files := \varnothing$
act7 : $r\_opened\_files := \varnothing$
act8 : $wbuffer := \varnothing$
act9 : $rbuffer := \varnothing$
act10 : $power\_on := TRUE$
end

Event *w_open* $\widehat{=}$

Open the given file for writing.

any
f
cnt
where
grd1 : $f \in files$
grd2 : $cnt \in CONTENT$
The content to be written.
grd3 : $f \notin w\_opened\_files \cup r\_opened\_files$
grd4 : $power\_on = TRUE$
then
act1 : $w\_opened\_files := w\_opened\_files \cup \{f\}$
act2 : $wbuffer(f) := cnt$
Set wbuffer pointing to the content to be written.
end

Event *writefile* $\widehat{=}$

Write the content on the writing buffer of the given file (*wbuffer(f)*) into the storage, start at the offset with the length (*len*) specified. The previous content of the file will be overridden by the content on the write buffer starting at the offset specified. The length to be written in the specification equals the length of the data on the write buffer

any
$f, offset, len,$
$corresPos$ mapping function between logical and physical page ids
where
grd1 : $f \in w\_opened\_files$
grd2 : $offset \in \mathbb{N}$

grd3 : $len \in \mathbb{N}$
grd4 : $len \leq card(wbuffer(f))$
grd5 : $corresPos \in 0\,..\,len-1 \rightarrowtail offset\,..\,offset+len-1$
grd6 : $\forall p \cdot p \in dom(corresPos) \Rightarrow corresPos(p) = p + offset$
grd7 : $power\_on = TRUE$

then

act1 : $fcontent(f) := fcontent(f) \lhd\!\!\!- (corresPos^{-1};\ (0\,..\,len-1 \lhd wbuffer(f)))$

end

Event *r_open* $\widehat{=}$

Open file for reading

any

f

where

grd1 : $f \in files$
grd2 : $f \notin w\_opened\_files \cup r\_opened\_files$
grd3 : $power\_on = TRUE$

then

act1 : $r\_opened\_files := r\_opened\_files \cup \{f\}$
act2 : $rbuffer(f) := \varnothing$

end

Event *readfile* $\widehat{=}$

Read the content of file f from the storage, starting at the offset with the length (len) specified, into the read buffer.

any

$f, offset, len$

where

grd1 : $power\_on = TRUE$
grd2 : $f \in r\_opened\_files$
grd3 : $offset \in dom(fcontent(f))$
grd4 : $len \in \mathbb{N}$
grd5 : $offset + len - 1 \in dom(fcontent(f))$

then

act1 : $rbuffer(f) := (offset\,..\,offset + len - 1) \lhd fcontent(f)$

end

Event *close* $\widehat{=}$

Close an opened file.

any

f

where

grd1 : $f \in r\_opened\_files \cup w\_opened\_files$
grd2 : $power\_on = TRUE$

then

act1 : $r\_opened\_files := r\_opened\_files \setminus \{f\}$
act2 : $w\_opened\_files := w\_opened\_files \setminus \{f\}$
act3 : $rbuffer := \{f\} \lhd\!\!\!- rbuffer$
act4 : $wbuffer := \{f\} \lhd\!\!\!- wbuffer$

end

END

B.2 The fifth refinement: Decomposition of the write event

MACHINE FMCH05B

REFINES FMCH04B

SEES FCTX03

VARIABLES

...

writing files being in the writing state

fcont_tmp temporary contents of writing files

writing_offset the given offsets to be written of writing files

writing_len the specified lengths to be written of writing files

INVARIANTS

inv1 : $writing \subseteq w\_opened\_files$

inv2 : $fcont\_tmp \in writing \rightarrow CONTENT$

inv3 : $\forall f \cdot f \in writing \Rightarrow fcont\_tmp(f) \subseteq wbuffer(f)$

inv4 : $writing\_offset \in writing \rightarrow \mathbb{N}$

inv5 : $writing\_len \in writing \rightarrow \mathbb{N}$

inv6 : $\forall f \cdot f \in writing \Rightarrow writing\_len(f) \leq card(wbuffer(f))$

inv7 : $\forall f \cdot f \in writing \Rightarrow writing\_offset(f) \in dom(fcontent(f))$

inv8 : $power\_on = FALSE \Rightarrow (writing = \varnothing \wedge fcont\_tmp = \varnothing)$

EVENTS

Initialisation

extended

begin

act20 : $writing := \varnothing$

act21 : $fcont\_tmp := \varnothing$

act34 : $writing\_offset := \varnothing$

act35 : $writing\_len := \varnothing$

end

Event $w\_open \mathrel{\widehat{=}}$

Open file f for writing by user *usr* where *cnt* is the content to be written (on write buffer)

extends *w_open*

any

f, cnt, usr

where

grd1 : $\texttt{f} \in \texttt{files}$

grd2 : $\texttt{cnt} \in \texttt{CONTENT}$

The content to be written.

grd3 : $\texttt{f} \notin \texttt{w\_opened\_files} \cup \texttt{r\_opened\_files}$

grd6 : $\texttt{power\_on} = \texttt{TRUE}$

grd4 : $\texttt{usr} \in \texttt{users}$

grd5 : $\texttt{f} \mapsto \texttt{usr} \in \texttt{WPerm}(\texttt{obj\_perms} \mapsto \texttt{obj\_owner} \mapsto \texttt{obj\_grp} \mapsto \texttt{user\_grps})$

then

act1 : $\texttt{w\_opened\_files} := \texttt{w\_opened\_files} \cup \{\texttt{f}\}$

act2 : $\texttt{wbuffer}(\texttt{f}) := \texttt{cnt}$

Set wbuffer pointing to the content to be written.

end

Event *w_start* $\widehat{=}$

Start write. Specifies the offset and length (*len*) to be written. Sets f into the writing state.

any
$f, offset, len$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in w\_opened\_files$
grd3 : $f \notin writing$
grd4 : $offset \in \mathbb{N}$
grd5 : $len \in \mathbb{N}$
grd6 : $len \leq card(wbuffer(f))$
grd7 : $offset \in dom(fcontent(f))$
then
act1 : $writing := writing \cup \{f\}$
act2 : $fcont\_tmp(f) := \varnothing$
act3 : $writing\_offset(f) := offset$
act4 : $writing\_len(f) := len$
end

Event *w_step* $\widehat{=}$

Writing step, write the data of page i on the buffer into $fcont\_tmp(f)$ (which is a mirror content of f in the storage)

any
$f, i, data$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in writing$
grd3 : $i \in \mathbb{N}$
grd4 : $data \in DATA$
grd5 : $i \mapsto data \in wbuffer(f)$
grd6 : $i \notin dom(fcont\_tmp(f))$
then
act1 : $fcont\_tmp(f) := fcont\_tmp(f) \cup \{i \mapsto data\}$
end

Event *w_end_ok* $\widehat{=}$

Write the content on the *wbuffer* of the given file f into the storage, starting at the offset with the length specified. The previous content of the file will be overridden by the content on the write buffer starting at the offset specified. The length to be written in the specification equals the length of the data on the write buffer.

refines *writefile*

any
$f, offset, len, fsz, corresPos$
where
grd3 : $power\_on = TRUE$
grd1 : $f \in writing$
grd2 : $offset = writing\_offset(f)$
grd4 : $len = writing\_len(f)$
grd5 : $corresPos \in 0 \,..\, len - 1 \rightarrowtail\!\!\!\!\!\!\rightarrow offset \,..\, offset + len - 1$
grd6 : $\forall p \cdot p \in dom(corresPos) \Rightarrow corresPos(p) = p + offset$
grd7 : $dom(fcont\_tmp(f)) = 0 \,..\, len - 1$
grd8 : $fsz \in \{len + offset, file\_size(f)\}$
grd9 : $fsz = len + offset \Leftrightarrow offset + len > file\_size(f)$
then

act1 : $fcontent(f) := fcontent(f) \lhd (corresPos^{-1};\ fcont\_tmp(f))$
act2 : $dateLastModified(f) := nowdate$
act3 : $file\_size(f) := fsz$
act4 : $writing := writing \setminus \{f\}$
act5 : $fcont\_tmp := \{f\} \lhd fcont\_tmp$
act6 : $writing\_offset := \{f\} \lhd writing\_offset$
act7 : $writing\_len := \{f\} \lhd writing\_len$

end

Event $w\_end\_fail \mathrel{\widehat{=}}$

write fail

any
f
where
grd1 : $f \in writing$
then
act2 : $writing := writing \setminus \{f\}$
act3 : $fcont\_tmp := \{f\} \lhd fcont\_tmp$
act4 : $writing\_offset := \{f\} \lhd writing\_offset$
act5 : $writing\_len := \{f\} \lhd writing\_len$
end

END

B.3 The sixth refinement: Decomposition of the read event

MACHINE FMCH06B

Decompose the file write event

REFINES FMCH05B

SEES FCTX03

VARIABLES

...

$reading$ files being read

$rbuff\_tmp$ temporary read-buffers of reading files

$reading\_offset$ the offset to be started of reading files

$reading\_len$ the length to be read

INVARIANTS

inv1 : $reading \subseteq r\_opened\_files$

inv2 : $rbuff\_tmp \in reading \rightarrow CONTENT$

inv3 : $reading\_offset \in reading \rightarrow \mathbb{N}$

inv4 : $reading\_len \in reading \rightarrow \mathbb{N}$

inv5 : $\forall f \cdot f \in reading \wedge power\_on = TRUE \Rightarrow reading\_offset(f) \in dom(fcontent(f))$

inv6 : $\forall f \cdot f \in reading \wedge power\_on = TRUE \Rightarrow reading\_offset(f) + reading\_len(f) - 1 \in dom(fcontent(f))$

inv7 : $\forall f \cdot f \in reading \wedge power\_on = TRUE \Rightarrow rbuff\_tmp(f) \subseteq reading\_offset(f) \mathinner{..} (reading\_offset(f) + reading\_len(f) - 1) \lhd fcontent(f)$

inv8 : $power\_on = FALSE \Rightarrow (reading = \varnothing \wedge rbuff\_tmp = \varnothing)$

EVENTS

Initialisation

extended

begin

act25 : $reading := \varnothing$

act26 : $rbuff\_tmp := \varnothing$

act27 : $reading\_offset := \varnothing$

act28 : $reading\_len := \varnothing$

end

Event $r\_open \mathrel{\widehat{=}}$

Open file f for reading, by user usr

extends r_open

any

f, usr

where

grd1 : $\mathtt{f} \in \mathtt{files}$

grd2 : $\mathtt{f} \notin \mathtt{w\_opened\_files} \cup \mathtt{r\_opened\_files}$

grd3 : $\mathtt{power\_on} = \mathtt{TRUE}$

grd4 : $\mathtt{usr} \in \mathtt{users}$

grd5 : $\mathtt{f} \mapsto \mathtt{usr} \in \mathtt{RPerm}(\mathtt{obj\_perms} \mapsto \mathtt{obj\_owner} \mapsto \mathtt{obj\_grp} \mapsto \mathtt{user\_grps})$

then

act1 : $\mathtt{r\_opened\_files} := \mathtt{r\_opened\_files} \cup \{\mathtt{f}\}$

act2 : $\mathtt{rbuffer}(\mathtt{f}) := \varnothing$

end

Event *r_start* $\widehat{=}$

Start read the given file at the offset with the length *len*.

any
$f, offset, len$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in r\_opened\_files$
grd3 : $f \notin reading$
grd4 : $offset \in \mathbb{N}$
grd5 : $offset \in 0 \mathinner{..} file\_size(f)$
grd6 : $len \in \mathbb{N}$
grd7 : $offset + len - 1 \in dom(fcontent(f))$
then
act1 : $reading := reading \cup \{f\}$
act2 : $rbuff\_tmp(f) := \varnothing$
act3 : $reading\_offset(f) := offset$
act4 : $reading\_len(f) := len$
end

Event *r_step* $\widehat{=}$

Reading step, read the data of page i from the storage into the temp read-buffer.

any
$f, i, data$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in reading$
grd3 : $i \in \mathbb{N}$
grd4 : $i \in reading\_offset(f) \mathinner{..} reading\_offset(f) + reading\_len(f) - 1$
grd5 : $data \in DATA$
grd6 : $i \mapsto data \in fcontent(f)$
grd7 : $i \notin dom(rbuff\_tmp(f))$
then
act1 : $rbuff\_tmp(f) := rbuff\_tmp(f) \cup \{i \mapsto data\}$
end

Event *r_end_ok* $\widehat{=}$

Reading file is succeeded, when all pages required have beeb read into the temp read-buffer (*grd*5).

refines *readfile*

any
$f, offset, len$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in reading$
grd3 : $offset = reading\_offset(f)$
grd4 : $len = reading\_len(f)$
grd5 : $dom(rbuff\_tmp(f)) = (offset \mathinner{..} offset + len - 1)$
then
act1 : $rbuffer(f) := rbuff\_tmp(f)$
act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$
act3 : $reading\_offset := \{f\} \lhd reading\_offset$
act4 : $reading\_len := \{f\} \lhd reading\_len$
act5 : $reading := reading \setminus \{f\}$
end

Event *r_end_fail* $\widehat{=}$

Reading file failed (abort). Release all buffer contents.

any
f
where
grd1 : $f \in reading$
then
act1 : $reading := reading \setminus \{f\}$
act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$
act3 : $reading\_offset := \{f\} \lhd reading\_offset$
act4 : $reading\_len := \{f\} \lhd reading\_len$
end
END

B.4 The seventh refinement: Introduction of the flash specification

This section outlines part of the specification that have beeb affected when the flash specification has been added.

MACHINE FMCH07B
REFINES FMCH06B
SEES FLCTX
VARIABLES

...
$flash$
$programmed\_pages$
$obsolete\_pages$
fat FAT table representing the table of content of each file
fat_tmp temporary FAT
$writing\_version$ writing version of each file
$most\_recent\_version$ the most recent version of file contents

INVARIANTS

inv1 : $power\_on = TRUE \Rightarrow flash \in RowAddr \rightarrow PDATA$
inv2 : $programmed\_pages \subseteq RowAddr$
inv3 : $obsolete\_pages \subseteq programmed\_pages$
inv4 : $power\_on = TRUE \Rightarrow fat \in files \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
inv5 : $fat\_tmp \in writing \rightarrow (\mathbb{N} \nrightarrow RowAddr)$
inv6 : $writing\_version \in writing \rightarrow VERNUM$
inv7 : $\forall f \cdot f \in files \wedge power\_on = TRUE \Rightarrow dom(fat(f)) = dom(fcontent(f))$
inv8 : $\forall f \cdot f \in writing \Rightarrow dom(fat\_tmp(f)) = dom(fcont\_tmp(f))$
inv9 : $\forall p \cdot p \in PDATA$
$\wedge\ objOfpage(p) \in dom(fat)$
$\wedge\ (\forall x \cdot x \in PDATA \wedge objOfpage(x) = objOfpage(p)$
$\wedge\ pidxOfpage(x) = pidxOfpage(p)$
$\Rightarrow\ verOfpage(x) < verOfpage(p)$
$)$
$\Rightarrow$
$pidxOfpage(p) \mapsto dataOfpage(p) \in fcontent(objOfpage(p))$
inv10 : $\forall p \cdot p \in PDATA \wedge objOfpage(p) \in dom(fat\_tmp)$
$\wedge\ verOfpage(p) = writing\_version(objOfpage(p))$
$\Rightarrow$
$pidxOfpage(p) \mapsto dataOfpage(p) \in wbuffer(objOfpage(p))$
inv11 : $\forall p \cdot p \in PDATA \wedge objOfpage(p) \in writing$
$\Rightarrow$
$verOfpage(p) \leq writing\_version(objOfpage(p))$
inv12 : $most\_recent\_version \in files \rightarrow VERNUM$

inv13 : $\forall p \cdot p \in PDATA \land objOfpage(p) \in writing$
$\Rightarrow$
$writing\_version(objOfpage(p)) > most\_recent\_version(objOfpage(p))$

inv13 : $power\_on = FALSE \Rightarrow (fat = \varnothing \land fat\_tmp = \varnothing \land writing\_version = \varnothing)$

inv14 : $\forall i, r, f, p \cdot f \in files \land r \in programmed\_pages \setminus obsolete\_pages$
$\land\ p = flash(r) \land objOfpage(p) = f \land pidxOfpage(p) = i \land i \neq 0$
$\land\ (\ \forall x \cdot x \in PDATA \land objOfpage(x) = f$
$\land\ pidxOfpage(x) = i$
$\Rightarrow\ verOfpage(x) < verOfpage(p)\ )$
$\Rightarrow\ i \mapsto r \in fat(f)$

inv15 : $\forall i, r, f, p \cdot r \in programmed\_pages \setminus obsolete\_pages \land f \in writing$
$\land\ p = flash(r) \land objOfpage(p) = f \land pidxOfpage(p) = i \land i \neq 0$
$\land\ (\ \forall x \cdot x \in PDATA \land objOfpage(x) = f$
$\land\ pidxOfpage(x) = i$
$\Rightarrow\ verOfpage(x) < verOfpage(p)\ )$
$\Rightarrow\ i \mapsto r \in fat\_tmp(f)$

inv16 : $\forall f \cdot f \in files \Rightarrow dom(fcontent(f)) = 1 \,..\, file\_size(f)$

EVENTS

Initialisation

begin

act1 : $files := \varnothing$
act2 : $directories := \{root\}$
act3 : $parent := \varnothing$
act4 : $w\_opened\_files := \varnothing$
act5 : $r\_opened\_files := \varnothing$
act6 : $wbuffer := \varnothing$
act7 : $rbuffer := \varnothing$
act8 : $users := \{su\}$
act9 : $groups := \{admin\}$
act10 : $user\_grps := \{su \mapsto admin\}$
act11 : $user\_pgrp := \{su \mapsto admin\}$
act12 : $obj\_owner := \{root \mapsto su\}$
act13 : $obj\_grp := \{root \mapsto admin\}$
act14 : $obj\_perms := \{root \mapsto wbo, root \mapsto rbo, root \mapsto xbo\}$
act15 : $oname := \{root \mapsto rname\}$
act16 : $dateCreated := \{root \mapsto dfdate\}$
act17 : $dateLastModified := \{root \mapsto dfdate\}$
act18 : $file\_size := \varnothing$
act19 : $power\_on := TRUE$
act20 : $writing := \varnothing$
act21 : $writing\_offset := \varnothing$
act22 : $writing\_len := \varnothing$
act23 : $reading := \varnothing$
act24 : $rbuff\_tmp := \varnothing$
act25 : $reading\_offset := \varnothing$
act26 : $reading\_len := \varnothing$
act27 : $flash := dflash$
act28 : $programmed\_pages := \varnothing$
act29 : $obsolete\_pages := \varnothing$
act30 : $fat := \varnothing$
act31 : $fat\_tmp := \varnothing$

act32 : $writing\_version := \varnothing$
act33 : $most\_recent\_version := \varnothing$

end

Event *mount* $\widehat{=}$

refines *mount*

any

$objs, fs, ds, prt, x, fcnt, objperms$
$objgrp, objname, cdate, mdate, fsize$
ft fat table of each file being mounted
mrv the most recent version of each file

where

grd1 : $power\_on = TRUE$
grd2 : $objs \subseteq OBJECT$
grd3 : $fs \subseteq objs$
grd4 : $ds \subseteq objs$
grd5 : $objs = fs \cup ds$
grd6 : $fs \cap ds = \varnothing$
grd7 : $(files \cup directories) \cap objs = \varnothing$
grd8 : $x \in ds$
grd9 : $prt \in objs \setminus \{x\} \to ds$
grd10 : $\forall s \cdot (s \subseteq prt^{-1}[s] \Rightarrow s = \varnothing)$
grd11 : $prt \cap parent = \varnothing$
grd12 : $files \cap fs = \varnothing$
grd13 : $directories \cap ds = \varnothing$
grd14 : $fcnt \in fs \to CONTENT$
grd15 : $objown \in objs \to users$
grd16 : $objperms \in objs \leftrightarrow PERMISSION$
grd17 : $objgrp \in objs \to groups$
grd18 : $objname \in objs \to NAME$
grd19 : $cdate \in objs \to DATE$
grd20 : $mdate \in objs \to DATE$
grd21 : $fsize \in fs \to \mathbb{N}$
grd22 : $ft \in fs \to (\mathbb{N} \nrightarrow RowAddr)$
grd23 : $\forall f \cdot f \in fs \Rightarrow dom(ft(f)) = dom(fcnt(f))$
grd24 : $\forall p \cdot p \in ran(flash)$

$$\begin{array}{l} \quad \wedge\ objOfpage(p) \in dom(ft) \\ \quad \wedge\ (\forall x \cdot x \in PDATA \wedge objOfpage(x) = objOfpage(p) \\ \qquad \wedge\ pidxOfpage(x) = pidxOfpage(p) \\ \qquad \Rightarrow verOfpage(x) < verOfpage(p) \\ \quad) \\ \quad \Rightarrow \\ pidxOfpage(p) \mapsto dataOfpage(p) \in fcnt(objOfpage(p)) \end{array}$$

grd25 : $mrv \in objs \to VERNUM$
grd26 : $\forall p \cdot p \in PDATA$

$$\begin{array}{l} \quad \wedge\ objOfpage(p) \in dom(ft) \\ \quad \Rightarrow \\ verOfpage(p) \leq mrv(objOfpage(p)) \end{array}$$

then

act1 : $files := files \cup fs$
act2 : $directories := directories \cup ds$
act3 : $parent := parent \cup prt \cup \{x \mapsto root\}$

act4 : $fat := fat \cup ft$
act5 : $obj\_owner := obj\_owner \cup objown$
act6 : $obj\_perms := obj\_perms \cup objperms$
act7 : $obj\_grp := obj\_grp \cup objgrp$
act8 : $oname := oname \cup objname$
act9 : $dateCreated := dateCreated \cup cdate$
act10 : $dateLastModified := dateLastModified \cup mdate$
act11 : $file\_size := file\_size \cup fsize$
act12 : $most\_recent\_version := most\_recent\_version \cup mrv$

end

Event *unmount* $\widehat{=}$

refines *unmount*

any
$objs, x$
where
grd4 : $power\_on = TRUE$
grd1 : $objs \subseteq files \cup directories$
grd2 : $root \notin objs$
grd3 : $x \in objs$
grd5 : $objs = (tcl(parent))^{-1}[\{x\}] \cup \{x\}$
grd6 : $objs \cap w\_opened\_files = \varnothing$
grd7 : $objs \cap r\_opened\_files = \varnothing$
then
act1 : $files := files \setminus (objs \cap files)$
act3 : $directories := directories \setminus (objs \cap directories)$
act2 : $parent := objs \lhd parent$
act4 : $fat := objs \lhd fat$
act5 : $obj\_owner := objs \lhd obj\_owner$
act6 : $obj\_grp := objs \lhd obj\_grp$
act7 : $obj\_perms := objs \lhd obj\_perms$
act8 : $oname := objs \lhd oname$
act9 : $dateCreated := objs \lhd dateCreated$
act10 : $dateLastModified := objs \lhd dateLastModified$
act11 : $file\_size := objs \lhd file\_size$
act12 : $most\_recent\_version := objs \lhd most\_recent\_version$

end

Event *w_start* $\widehat{=}$

refines *w_start*

any
$f, offset, len$
wv writing version of f
where
grd1 : $power\_on = TRUE$
grd2 : $f \in w\_opened\_files$
grd3 : $f \notin writing$
grd4 : $wv \in VERNUM$
grd5 : $wv = most\_recent\_version(f) + 1$
grd6 : $\forall p \cdot p \in PDATA \wedge objOfpage(p) = f \Rightarrow verOfpage(p) < wv$
grd7 : $offset \in \mathbb{N}$
grd8 : $len \in \mathbb{N}$
grd9 : $len \leq card(wbuffer(f))$
grd10 : $offset \in 0 \,..\, file\_size(f))$
then
act1 : $writing := writing \cup \{f\}$

act2 : $fat\_tmp(f) := \varnothing$
act3 : $writing\_version(f) := wv$
act4 : $writing\_offset(f) := offset$
act5 : $writing\_len(f) := len$

end

Event $w\_step \mathrel{\widehat{=}}$

refines w_step

any

$f, i, data$
r a row address, a location for writing data of page i
pd page data to be programmed to the flash
wv writing version

where

grd1 : $power\_on = TRUE$
grd2 : $f \in writing$
grd3 : $i \in \mathbb{N}$
grd4 : $data \in DATA$
grd5 : $i \mapsto data \in wbuffer(f)$
grd6 : $i \notin dom(fat\_tmp(f))$
grd7 : $r \in RowAddr$
grd8 : $r \notin programmed\_pages$
grd9 : $wv = writing\_version(f)$
grd10 : $pd \in PDATA$
grd11 : $objOfpage(pd) = f$
grd12 : $pidxOfpage(pd) = i$
grd13 : $verOfpage(pd) = wv$
grd14 : $dataOfpage(pd) = data$

then

act1 : $fat\_tmp(f) := fat\_tmp(f) \cup \{i \mapsto r\}$
act2 : $flash(r) := pd$
act3 : $programmed\_pages := programmed\_pages \cup \{r\}$

end

Event $w\_end\_ok \mathrel{\widehat{=}}$

Writing the content on the wbuffer of the given file into the storage. It is completed when all pages required have been programmed to the flash device ($grd10$).

refines w_end_ok

any

$f, offset, len, fsz, cnt, corresPos$

where

grd1 : $power\_on = TRUE$
grd2 : $f \in writing$
grd3 : $offset = writing\_offset(f)$
grd4 : $len = writing\_len(f)$
grd5 : $cnt \in \mathbb{N} \nrightarrow RowAddr$
grd6 : $offset \in dom(fat(f))$
grd7 : $corresPos \in 0 \mathinner{..} len - 1 \rightarrowtail offset \mathinner{..} offset + len - 1$
grd8 : $\forall p \cdot p \in dom(corresPos) \Rightarrow corresPos(p) = p + offset$
grd9 : $fsz \in \{len + offset, file\_size(f)\}$
grd10 : $fsz = len + offset \Leftrightarrow offset + len > file\_size(f)$
grd11 : $dom(fat\_tmp(f)) = 0 \mathinner{..} len - 1$
grd12 : $cnt = corresPos^{-1};\ fat\_tmp(f)$

then

act1 : $fat(f) := fat(f) \lhd\!\!\!- cnt$
act2 : $dateLastModified(f) := nowdate$

act3 : $file\_size(f) := fsz$
act4 : $fat\_tmp := \{f\} \lhd fat\_tmp$
act5 : $writing\_version := \{f\} \lhd writing\_version$
act6 : $writing := writing \setminus \{f\}$
act7 : $most\_recent\_version(f) := writing\_version(f)$
act8 : $writing\_offset := \{f\} \lhd writing\_offset$
act9 : $writing\_len := \{f\} \lhd writing\_len$

end

Event *w_end_fail* $\widehat{=}$

Writing the given file fails (abort). Releases all related memory contents.

refines *w_end_fail*

any
f
where
grd1 : $f \in writing$
then
act1 : $writing := writing \setminus \{f\}$
act2 : $fat\_tmp := \{f\} \lhd fat\_tmp$
act3 : $writing\_version := \{f\} \lhd writing\_version$
act4 : $writing\_offset := \{f\} \lhd writing\_offset$
act5 : $writing\_len := \{f\} \lhd writing\_len$
end

Event *r_start* $\widehat{=}$

Start read the given file f, starting at the offset with the length specified.

refines *r_start*

any
$f, offset, len$
where
grd1 : $power\_on = TRUE$
grd2 : $f \in r\_opened\_files$
grd3 : $f \notin reading$
grd4 : $offset \in \mathbb{N}$
grd5 : $offset \in 0 \mathinner{..} file\_size(f))$
grd6 : $len \in \mathbb{N}$
grd7 : $len \leq file\_size(f)$
then
act1 : $reading := reading \cup \{f\}$
act2 : $rbuff\_tmp(f) := \varnothing$
act3 : $reading\_offset(f) := offset$
act4 : $reading\_len(f) := len$
end

Event *r_step* $\widehat{=}$

Reading step, read the data (*data*) of page i from the storage (at row r) into the temp buffer.

refines *r_step*

any
$f, i, data, r$
pd page data of row r
ver the most recent version of page i
where
grd1 : $power\_on = TRUE$
grd2 : $f \in reading$
grd3 : $i \in \mathbb{N}$

grd4 : $i \in reading\_offset(f) \mathinner{..} reading\_offset(f) + reading\_len(f) - 1$
grd5 : $data \in DATA$
grd6 : $i \in dom(fat(f))$
grd7 : $r \in programmed\_pages \setminus obsolete\_pages$
grd8 : $r = fat(f)(i)$
grd9 : $i \notin dom(rbuff\_tmp(f))$
grd10 : $pd = flash(r)$
grd11 : $data = dataOfpage(pd)$
grd12 : $i = pidxOfpage(pd)$
grd13 : $f = objOfpage(pd)$
grd14 : $ver = verOfpage(pd)$

then

act1 : $rbuff\_tmp(f) := rbuff\_tmp(f) \cup \{i \mapsto data\}$

end

Event $r\_end\_ok \mathrel{\widehat{=}}$

Reading the given file end when all pages required have been read.

refines r_end_ok

any

$f, offset, len$

where

grd1 : $power\_on = TRUE$
grd2 : $f \in reading$
grd3 : $offset = reading\_offset(f)$
grd4 : $len = reading\_len(f)$
grd5 : $dom(rbuff\_tmp(f)) = (offset \mathinner{..} offset + len - 1)$

then

act1 : $rbuffer(f) := rbuff\_tmp(f)$
act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$
act3 : $reading\_offset := \{f\} \lhd reading\_offset$
act4 : $reading\_len := \{f\} \lhd reading\_len$
act5 : $reading := reading \setminus \{f\}$

end

Event $r\_end\_fail \mathrel{\widehat{=}}$

Read the whole content of a file from the storage into the read buffer.

refines r_end_fail

any

f

where

grd1 : $f \in reading$

then

act1 : $reading := reading \setminus \{f\}$
act2 : $rbuff\_tmp := \{f\} \lhd rbuff\_tmp$
act3 : $reading\_offset := \{f\} \lhd reading\_offset$
act4 : $reading\_len := \{f\} \lhd reading\_len$

end

Event $power\_loss \mathrel{\widehat{=}}$

refines $power\_loss$

when

grd1 : $power\_on = TRUE$

then

act1 : $power\_on := FALSE$
act2 : $w\_opened\_files := \varnothing$
act3 : $r\_opened\_files := \varnothing$
act4 : $wbuffer := \varnothing$

act5 : $rbuffer := \varnothing$
act6 : $writing := \varnothing$
act7 : $fat\_tmp := \varnothing$
act8 : $writing\_offset := \varnothing$
act9 : $writing\_len := \varnothing$
act10 : $reading := \varnothing$
act11 : $rbuff\_tmp := \varnothing$
act12 : $reading\_offset := \varnothing$
act13 : $reading\_len := \varnothing$
act14 : $writing\_version := \varnothing$

end

Event $power\_on \mathrel{\widehat{=}}$

refines *power_on*

any
ft

where
grd1 : power_on = FALSE
grd2 : $ft \in files \to (\mathbb{N} \nrightarrow RowAddr)$
grd3 : $\forall f \cdot f \in files \Rightarrow dom(ft(f)) = 1 \mathinner{..} file\_size(f)$
grd4 : $\forall p \cdot p \in PDATA \land objOfpage(p) \in dom(ft) \Rightarrow p \in ran(flash)$
grd5 : $\forall i, r, f, p \cdot f \in files \land r \in programmed\_pages \setminus obsolete\_pages$
$\land\ p = flash(r) \land objOfpage(p) = f \land pidxOfpage(p) = i \land i \neq 0$
$\land\ (\ \forall x \cdot x \in PDATA \land objOfpage(x) = f$
$\land\ pidxOfpage(x) = i$
$\Rightarrow\ verOfpage(x) < verOfpage(p)\ )$
$\Rightarrow\ i \mapsto r \in ft(f)$
grd6 : $\forall i, r, f, p \cdot f \in files \land r \in programmed\_pages \setminus obsolete\_pages$
$\land\ p = flash(r) \land i \mapsto r \in ft(f)$
$\Rightarrow\ (\ verOfpage(p) < most\_recent\_version(f) \land$
$objOfpage(p) = f \land pidxOfpage(p) = i\ )$

then
act1 : power_on := TRUE
act2 : $fat := ft$

end

END

Appendix C

An Event-B Specification of Flash Memory

C.1 An initial model

MACHINE FMCH07_FL

The flash part after decomposing.

SEES FLCTX

VARIABLES

$flash$ represents the flash device which is an array of page data

$programmed\_pages$ set of pages that have been programmed

$obsolete\_pages$ set of obsolete pages

INVARIANTS

inv1 : $flash \in RowAddr \rightarrow PDATA$

inv2 : $programmed\_pages \subseteq RowAddr$

inv3 : $obsolete\_pages \subseteq programmed\_pages$

EVENTS

Initialisation

begin

act1 : $flash := dflash$

act2 : $programmed\_pages := \varnothing$

act3 : $obsolete\_pages := \varnothing$

end

Event $page\_programme \mathrel{\widehat{=}}$

Programme data to the flash at the given row address.

any

new_r

$pdata$

where

grd1 : $new\_r \in RowAddr \setminus programmed\_pages$

grd2 : $pdata \in PDATA$

then

act1 : $flash(new\_r) := pdata$
act2 : $programmed\_pages := programmed\_pages \cup \{new\_r\}$
end

Event *page_read* $\widehat{=}$

Read page data from the flash at the given row address.

any
r
$pdata$
where
grd1 : $r \in programmed\_pages \setminus obsolete\_pages$
grd2 : $pdata = flash(r)$
end

Event *block_erase* $\widehat{=}$

Erase all the given pages within a block.

any
$rows$ All rows within the given block.
where
grd1 : $rows \subseteq RowAddr$
grd2 : $rows \cap (programmed\_pages \setminus obsolete\_pages) = \varnothing$
The block being erased have no vailid pages
then
act1 : $flash := flash \Leftarrow (rows \times \{dp\})$
act2 : $programmed\_pages := programmed\_pages \setminus rows$
act3 : $obsolete\_pages := obsolete\_pages \setminus rows$
end

Event *mark_pages_obsolete* $\widehat{=}$

Utility event. Mark all pages belongs to *obj* as obsolete.

any
$rows$
obj
where
grd1 : $obj \in OBJECT$
grd2 : $rows \subseteq programmed\_pages \setminus obsolete\_pages$
grd3 : $rows = flash^{-1}[objOfpage^{-1}[\{obj\}]]$
then
act1 : $obsolete\_pages := obsolete\_pages \cup rows$
end

Event *mark_a_page_obsolete* $\widehat{=}$

Utility event. Marks a page specified by the given row address as obsolete.

any
old_r
where
grd1 : $old\_r \in programmed\_pages \setminus obsolete\_pages$
then
act1 : $obsolete\_pages := obsolete\_pages \cup \{old\_r\}$
end

END

C.2 The first refinement: Page Register

Page registers are introduced in this step. Two phases are required for the events *page_read* and *page_write*.

MACHINE FMCH07_FL_REF1

REFINES FMCH07_FL

SEES FLCTX01

VARIABLES

`flash`

`programmed_pages`

`obsolete_pages`

$ready2read$ Set of page registers that is content is ready to be read off chip.

$ready2prog$ Set of page registers that is content is ready to be programmed.

$readingPR$ Set of page registers being in the reading state.

$writingPR$ Set of page registers being in the writing state.

$corresRowOfreadingPR$ the corresponding row address of the page register being used for reading

$corresRowOfwritingPR$ the corresponding row address of the page register being used for writing

$dataOfPR$ the page data within the page register

INVARIANTS

`inv1` : $readingPR \subseteq PR$

`inv2` : $writingPR \subseteq PR$

`inv3` : $readingPR \cap writingPR = \varnothing$

`inv4` : $ready2read \subseteq readingPR$

`inv5` : $ready2prog \subseteq writingPR$

`inv6` : $dataOfPR \in PR \rightarrow PDATA$

`inv7` : $corresRowOfreadingPR \in readingPR \rightarrow programmed\_pages \setminus obsolete\_pages$

`inv8` : $corresRowOfwritingPR \in writingPR \rightarrowtail RowAddr \setminus programmed\_pages$

`inv9` : $\forall pr, r \cdot pr \in ready2read \land r \in programmed\_pages$
$\land\ r = corresRowOfreadingPR(pr)$
$\Rightarrow dataOfPR(pr) = flash(r)$

`inv10` : $ran(corresRowOfreadingPR) \cap ran(corresRowOfwritingPR) = \varnothing$

EVENTS

Initialisation

extended

begin

`act4` : $ready2read := \varnothing$

`act5` : $ready2prog := \varnothing$

`act6` : $readingPR := \varnothing$

`act7` : $writingPR := \varnothing$

`act8` : $corresRowOfreadingPR := \varnothing$

`act9` : $corresRowOfwritingPR := \varnothing$

`act10` : $dataOfPR := PR \times \{dp\}$

end

Event *pageread_start* $\widehat{=}$

(read step1) Start reading a page by selecting the related LUN (in which the the page row r is) and available page register.

any

r the target row address to be read
lid LUN number to which the r belongs
pr an available page register within LUN lid

where

grd1 : $r \in programmed\_pages \setminus obsolete\_pages$
grd2 : $pr \in PR$
grd3 : $pr \notin readingPR \cup writingPR$
grd4 : $lid = lidOfPR(pr)$
grd5 : $lidOfRow(r) = lid$

then

act1 : $readingPR := readingPR \cup \{pr\}$
act2 : $corresRowOfreadingPR(pr) := r$

end

Event *read2reg* $\widehat{=}$

(read step2) Transfer the page-data ($pdata$) at the given row address (r) to the page register (pr).

any

$r, pdata, pr$

where

grd1 : $pr \in dom(corresRowOfreadingPR)$
grd2 : $r = corresRowOfreadingPR(pr)$
grd3 : $pr \in readingPR$
grd4 : $pr \notin ready2read$
grd5 : $pdata \in PDATA$
grd6 : $pdata = flash(r)$

then

act1 : $dataOfPR(pr) := pdata$
act2 : $ready2read := ready2read \cup \{pr\}$

end

Event *page_read_end* $\widehat{=}$

(read step3 success) Read page data ($pdata$) of row r from the register (pr) off chip.

refines *page_read*

any

$r, pdata, pr$

where

grd1 : $pr \in ready2read$
grd2 : $r = corresRowOfreadingPR(pr)$
grd3 : $pdata = dataOfPR(pr)$

then

act1 : $ready2read := ready2read \setminus \{pr\}$
act2 : $readingPR := readingPR \setminus \{pr\}$
act3 : $corresRowOfreadingPR := \{pr\} \lhd corresRowOfreadingPR$

end

Event *page_read_fail* $\widehat{=}$

(read step3 fails)

any

r
pr

where

grd1 : $pr \in readingPR$
grd2 : $r = corresRowOfreadingPR(pr)$

then

act1 : $readingPR := readingPR \setminus \{pr\}$
act2 : $ready2read := ready2read \setminus \{pr\}$
act3 : $corresRowOfreadingPR := \{pr\} \lhd corresRowOfreadingPR$

end

Event *pageprog_start* $\widehat{=}$

(program step1) Starting page_programmed by selecting the related LUN and avialable page register.

any

r the row address to which the data to be programmed
lid the LUN to which the row r belongs
pr an available page register within the LUN

where

grd1 : $r \in RowAddr \setminus programmed\_pages$
grd2 : $r \notin ran(corresRowOfwritingPR) \cup ran(corresRowOfwritingPR)$
grd3 : $pr \in PR$
grd4 : $pr \notin readingPR \cup writingPR$
grd5 : $lid = lidOfPR(pr)$
grd6 : $lidOfRow(r) = lid$

then

act1 : $writingPR := writingPR \cup \{pr\}$
act2 : $corresRowOfwritingPR(pr) := r$

end

Event *write2reg* $\widehat{=}$

(program step2) Write/transfer the page data *pdata* to be programmed into the page register (pr) within the LUN (lid).

any

$r, pdata, pr$

where

grd1 : $r \in RowAddr$
grd2 : $pr \in writingPR$
grd3 : $pr \notin ready2prog$
grd4 : $lidOfRow(r) = lidOfPR(pr)$
grd5 : $corresRowOfwritingPR(pr) = r$
grd6 : $pdata \in PDATA$

then

act1 : $dataOfPR(pr) := pdata$
act2 : $ready2prog := ready2prog \cup \{pr\}$

end

Event *page_program_ok* $\widehat{=}$

(program step3 successful) Program the data on the page register to the flash at the corresponding row address.

refines *page_programme*

any

$new\_r, pdata, pr$

where

grd1 : $pdata \in PDATA$
grd2 : $pr \in ready2prog$

grd3 : $new\_r = corresRowOfwritingPR(pr)$
grd4 : $pdata = dataOfPR(pr)$

then

act1 : $flash(new\_r) := pdata$
act2 : $programmed\_pages := programmed\_pages \cup \{new\_r\}$
act3 : $ready2prog := ready2prog \setminus \{pr\}$
act4 : $writingPR := writingPR \setminus \{pr\}$
act5 : $corresRowOfwritingPR := \{pr\} \lhd corresRowOfwritingPR$

end

Event *page_prog_fail* $\widehat{=}$

(program step3 failed)

any

r, pr

where

grd1 : $pr \in writingPR$
grd2 : $r = corresRowOfwritingPR(pr)$

then

act1 : $writingPR := writingPR \setminus \{pr\}$
act2 : $ready2prog := ready2prog \setminus \{pr\}$
act3 : $corresRowOfwritingPR := \{pr\} \lhd corresRowOfwritingPR$

end

Event *mark_pages_obsolete* $\widehat{=}$

A utility event used for marking a set of pages as obsolete.

extends *mark_pages_obsolete*

where

grd4 : $rows \cap (ran(corresRowOfreadingPR) \cup ran(corresRowOfwritingPR)) = \varnothing$

end

Event *mark_a_page_obsolete* $\widehat{=}$

Utility event. Marks a single page specified by the given row address as obsolete.

extends *mark_a_page_obsolete*

where

grd2 : $old\_r \notin ran(corresRowOfreadingPR) \cup ran(corresRowOfwritingPR)$

end

Event *erase_a_block* $\widehat{=}$

extends *block_erase*

where

grd3 : $rows \cap ran(corresRowOfreadingPR) = \varnothing$
grd4 : $rows \cap ran(corresRowOfwritingPR) = \varnothing$

end

END

C.3 The second refinement: Relocation events

Block reclamation is a background process. It is composed of (1) Selecting a candidate block to reclaim. In our development, we select an block with the least number of erasures. (2) Relocating if any valid page exists. After relocating complete, the given block becomes obsolete (ready to be erased). (3) Erasing an obsolete block to be reused. This process may not be proceed once the second step has been completed. We assume that the obsolete will be selected when free spaces are required or when the system is in the idle state (depending on algorithm). We delay the erase event to be refined in the fourth refinement, where more details of reclamation process are added.

Two events required for relocation a page are introduced in this refinement: (1) copy a valid page from one place to another fresh page (2) mark the old location to be obsolete.

MACHINE FMCH07_FL_REF2d
REFINES FMCH07_FL_REF1
SEES FLCTX01
VARIABLES

`flash`
`programmed_pages`
`obsolete_pages`
`ready2read`
`ready2prog`
`readingPR` Set of page registers being in the reading state.
`writingPR` Set of page registers being in the writing state.
`corresRowOfreadingPR`
`corresRowOfwritingPR`
`dataOfPR`
$flash2$ represents part of the flash array that have been programmed during relocating process.
$trans\_func$ A translation function, mapping the old location to the new location.

$programmed\_pages2$ represents a set of pages that have been programmed during the relocation.
$obsolete\_pages2$ represents a set of all obsolete pages.

INVARIANTS

inv1 : $flash2 \in RowAddr \pfun PDATA$
inv2 : $trans\_func \in RowAddr \pfun RowAddr$
inv3 : $programmed\_pages2 \subseteq RowAddr$
inv4 : $programmed\_pages2 = trans\_func[programmed\_pages]$
inv5 : $dom(flash2) = programmed\_pages2$
inv6 : $programmed\_pages \cap programmed\_pages2 = \varnothing$
inv7 : $obsolete\_pages2 \subseteq programmed\_pages \cup programmed\_pages2$
inv8 : $obsolete\_pages \subseteq obsolete\_pages2$

inv9 : $dom(trans\_func) \subseteq programmed\_pages$
inv10 : $ran(trans\_func) \cap obsolete\_pages2 = \varnothing$
inv11 : $\forall r \cdot r \in dom(trans\_func) \Rightarrow flash(r) = flash2(trans\_func(r))$
inv12 : $\forall r \cdot r \in ran(corresRowOfwritingPR) \Rightarrow r \notin programmed\_pages2$

EVENTS

Initialisation

extended

begin
act13 : $flash2 := dflash$
act14 : $trans\_func := \varnothing$
act15 : $programmed\_pages2 := \varnothing$
act16 : $obsolete\_pages2 := \varnothing$
end

Event *pageread_start* $\widehat{=}$

(read step1) Starting page_read by selecting the related LUN (*lid*) (in which the the page row r is) and available page register (pr).

extends *pageread_start*

any
r, lid, pr
where
grd1 : r ∈ programmed_pages \ obsolete_pages2
grd2 : pr ∈ PR
grd3 : pr ∉ readingPR ∪ writingPR
grd4 : lid = lidOfPR(pr)
grd5 : lidOfRow(r) = lid
then
act1 : readingPR := readingPR ∪ {pr}
act2 : corresRowOfreadingPR(pr) := r
end

Event *read2reg* $\widehat{=}$

(read step2) Transfer the page-data at the given row address (r) to the page register (pr). case1: row r has not been relocated

extends *read2reg*

any
r, pdata, pr
where
grd1 : pr ∈ dom(corresRowOfreadingPR)
grd2 : r = corresRowOfreadingPR(pr)
grd7 : $r \notin dom(trans\_func)$
grd3 : pr ∈ readingPR
grd4 : pr ∉ ready2read
grd5 : pdata ∈ PDATA
grd6 : pdata = flash(r)
then
act1 : dataOfPR(pr) := pdata
act2 : ready2read := ready2read ∪ {pr}
end

Event *read2reg2* $\widehat{=}$

(r2) Transfer the page-data (*pdata*) at the given row address to the page register pr. case2: if row r has been relocated. The content will be the content where the content of the given page has been relocated (*grd7*)

refines *read2reg*

any
$r, pdata, pr$
where
grd1 : $pr \in dom(corresRowOfreadingPR)$
grd2 : $r = corresRowOfreadingPR(pr)$
grd3 : $pr \in readingPR$
grd4 : $pr \notin ready2read$
grd5 : $pdata \in PDATA$
grd6 : $r \in dom(trans\_func)$
grd7 : $pdata = flash2(trans\_func(r))$
then
act1 : $dataOfPR(pr) := pdata$
act3 : $ready2read := ready2read \cup \{pr\}$
end

Event *page_read_end* $\widehat{=}$

(read step3 success) Read page data from the register off chip.

extends *page_read_end*

any
r, pdata, pr
where
grd1 : pr $\in$ ready2read
grd2 : r $=$ corresRowOfreadingPR(pr)
grd3 : pdata $=$ dataOfPR(pr)
then
act1 : ready2read $:=$ ready2read $\setminus$ {pr}
act2 : readingPR $:=$ readingPR $\setminus$ {pr}
act3 : corresRowOfreadingPR $:=$ {pr} $\lhd$ corresRowOfreadingPR
end

Event *page_read_fail* $\widehat{=}$

(read step3 fails)

extends *page_read_fail*

any
r, pr
where
grd1 : pr $\in$ readingPR
grd2 : r $=$ corresRowOfreadingPR(pr)
then
act1 : readingPR $:=$ readingPR $\setminus$ {pr}
act2 : ready2read $:=$ ready2read $\setminus$ {pr}
act3 : corresRowOfreadingPR $:=$ {pr} $\lhd$ corresRowOfreadingPR
end

Event *pageprog_start* $\widehat{=}$

(program step1) Starting page_programmed by selecting the related LUN *lid* (to which row r belongs) and available page register (pr).

refines *pageprog_start*

any
r, lid, pr
where
grd1 : $r \in RowAddr \setminus (programmed\_pages \cup programmed\_pages2)$
grd2 : $r \notin ran(corresRowOfwritingPR) \cup ran(corresRowOfwritingPR)$
grd3 : $pr \in PR$
grd4 : $pr \notin readingPR \cup writingPR$

grd5 : $lid = lidOfPR(pr)$
grd6 : $lidOfRow(r) = lid$

then

act1 : $writingPR := writingPR \cup \{pr\}$
act2 : $corresRowOfwritingPR(pr) := r$

end

Event *write2reg* $\widehat{=}$

(program step2) Write/transfer the data to be programmed into the page register within the LUN.

extends *write2reg*

any

r row address to be programmed
pdata
pr corresponding page register of row r

where

grd1 : $r \in RowAddr$
grd8 : $pr \in writingPR$
grd5 : $pr \notin ready2prog$
grd4 : $lidOfRow(r) = lidOfPR(pr)$
grd9 : $corresRowOfwritingPR(pr) = r$
grd6 : $pdata \in PDATA$

then

act1 : $dataOfPR(pr) := pdata$
act2 : $ready2prog := ready2prog \cup \{pr\}$

end

Event *page_program_ok* $\widehat{=}$

(end program success) Programme the data (*pdata*) on the page register pr to the flash at the corresponding row address r.

refines *page_program_ok*

any

$new\_r, pdata, pr$

where

grd1 : $pdata \in PDATA$
grd2 : $pr \in ready2prog$
grd3 : $new\_r = corresRowOfwritingPR(pr)$
grd4 : $pdata = dataOfPR(pr)$
grd5 : $new\_r \notin dom(trans\_func)$

then

act1 : $flash(new\_r) := pdata$
act2 : $programmed\_pages := programmed\_pages \cup \{new\_r\}$
act3 : $ready2prog := ready2prog \setminus \{pr\}$
act4 : $writingPR := writingPR \setminus \{pr\}$
act5 : $corresRowOfwritingPR := \{pr\} \lhd corresRowOfwritingPR$

end

Event *page_prog_fail* $\widehat{=}$

(programming a page fails)

extends *page_prog_fail*

any

r, pr

where

grd1 : $pr \in writingPR$
grd2 : $r = corresRowOfwritingPR(pr)$

then

act1 : writingPR := writingPR \ {pr}
act2 : ready2prog := ready2prog \ {pr}
act3 : corresRowOfwritingPR := {pr} ⩤ corresRowOfwritingPR

end

Event *mark_pages_obsolete* ≙

A utility event used for marking a set of pages (identified by *rows*) that belong to object *obj* as obsolete.

extends *mark_pages_obsolete*

any
rows, obj
where
grd1 : obj ∈ OBJECT
grd2 : rows ⊆ programmed_pages \ obsolete_pages
grd3 : rows = flash^{-1}[objOfpage^{-1}[{obj}]]
grd4 : rows∩(ran(corresRowOfreadingPR)∪ran(corresRowOfwritingPR)) = ∅
then
act1 : obsolete_pages := obsolete_pages ∪ rows
act2 : *obsolete_pages2* := *obsolete_pages2* ∪ *rows*
end

Event *mark_a_page_obsolete* ≙

Utility event. Marks a single page specified by the given row address as obsolete.

extends *mark_a_page_obsolete*

any
old_r
where
grd1 : old_r ∈ programmed_pages \ obsolete_pages
grd2 : old_r ∉ ran(corresRowOfreadingPR)∪ran(corresRowOfwritingPR)
then
act1 : obsolete_pages := obsolete_pages ∪ {old_r}
act2 : *obsolete_pages2* := *obsolete_pages2* ∪ {*old_r*}
end

Event *copy_a_page_to_new_loc* ≙

Copy a valid page from *old_r* to another location *new_r*

any
old_r, *new_r*, *pdata*
where
grd1 : *old_r* ∈ *programmed_pages* \ *obsolete_pages2*
grd2 : *new_r* ∈ *RowAddr* \ (*programmed_pages* ∪ *programmed_pages2*)
grd3 : *pdata* = *flash*(*old_r*)
grd4 : *old_r* ∉ *dom*(*trans_func*)
then
act1 : *flash2*(*new_r*) := *pdata*
act2 : *programmed_pages2* := *programmed_pages2* ∪ {*new_r*}
act3 : *trans_func*(*old_r*) := *new_r*
end

Event *mark_old_page_obsolete* ≙

Mark a page to be obsolete

any
old_r
where
grd1 : *old_r* ∈ *programmed_pages*
grd2 : *old_r* ∉ *obsolete_pages2*

then
act1 : $obsolete\_pages2 := obsolete\_pages2 \cup \{old\_r\}$
end

Event $erase\_a\_block \mathrel{\widehat{=}}$

Erase all the given pages within the given block, which is obsolete.

refines $erase\_a\_block$

any
$rows$ All rows within the given block.
where
grd1 : $rows \subseteq RowAddr$
grd2 : $rows \cap (programmed\_pagess \setminus obsolete\_pages2) = \varnothing$
grd3 : $rows \cap ran(corresRowOfwritingPR) = \varnothing$
grd4 : $rows \cap ran(corresRowOfreadingPR) = \varnothing$
grd5 : $rows \cap dom(trans\_func) = \varnothing$
grd2 : $rows \cap (programmed\_pagess2 \setminus obsolete\_pages2) = \varnothing$
then
act1 : $flash := flash \lhd\!\!\!- (rows \times \{dp\})$
act2 : $programmed\_pages := programmed\_pages \setminus rows$
act3 : $obsolete\_pages := obsolete\_pages \setminus rows$
act4 : $programmed\_pages2 := programmed\_pages2 \setminus rows$
act5 : $flash2 := flash2 \lhd\!\!\!- (rows \times \{dp\})$
act6 : $obsolete\_pages2 := obsolete\_pages2 \setminus rows$
end

END

C.4 The third refinement: Sequencing of relocation events

MACHINE FMCH07_FL_REF3

REFINES FMCH07_FL_REF2

SEES FLCTX2

VARIABLES

`flash`
`programmed_pages`
`obsolete_pages`
`ready2read`
`ready2prog`
`readingPR`
`writingPR`
`corresRowOfreadingPR`
`corresRowOfwritingPR`
`dataOfPR`
`flash2`
`trans_func`
`programmed_pages2`
`obsolete_pages2`
$relocating\_blocks$ a set of blocks being relocated
$relocating\_pages$ a set of pages being relocated from the old locations to new locations

INVARIANTS

`inv1` : $relocating\_blocks \subseteq BLOCK$
`inv2` : $relocating\_pages \in RowAddr \rightarrowtail RowAddr$
`inv3` : $dom(relocating\_pages) \subseteq (programmed\_pages2 \setminus obsolete\_pages2)$
`inv4` : $\forall b, r \cdot b \in relocating\_blocks \land r \in RowAddr$
$\land\ BlkOfRow(r) = b$
$\Rightarrow$
$r \notin ran(corresRowOfreadingPR) \land r \notin ran(corresRowOfwritingPR)$

EVENTS

Initialisation

extended

begin
`act17` : $relocating\_blocks := \varnothing$
`act18` : $relocating\_pages := \varnothing$
end

Event *pageread_start* $\widehat{=}$

(r1) Starting page_read by selecting the related LUN (in which the the page no. r is) and available page register.

extends *pageread_start*

where
`grd9` : $BlkOfRow(r) \notin relocating\_blocks$
end

Event *read2reg* $\widehat{=}$

(r2) Transfer the page-data at the given row address to the page register. Case1: the given row has not been relocated.

extends *read2reg*

Event *read2reg2* $\widehat{=}$

(r2) Transfer the page-data at the given row address to the page register. case2: if row r has been relocated.

extends *read2reg2*

Event *page_read_end* $\widehat{=}$

(r3a) Read page data from the register off chip.

extends *page_read_end*

Event *page_read_fail* $\widehat{=}$

(r3b)

extends *page_read_fail*

Event *pageprog_start* $\widehat{=}$

(w1) Starting page_programmed by selecting the related LUN and available page register.

extends *pageprog_start*

where

grd9 : $BlkOfRow(r) \notin relocating\_blocks$

end

Event *write2reg* $\widehat{=}$

(w2) Write/transfer the data to be programmed into the page register within the LUN.

extends *write2reg*

Event *page_program_ok* $\widehat{=}$

(w3a) Programme the data on the page register to the flash at the corresponding row address.

extends *page_program_ok*

Event *page_prog_fail* $\widehat{=}$

(w3b)

extends *page_prog_fail*

Event *mark_pages_obsolete* $\widehat{=}$

A utility event used for marking a set of pages as obsolete.

extends *mark_pages_obsolete*

where

grd5 : $rows \cap dom(relocating\_pages) = \varnothing$

end

Event *mark_a_page_obsolete* $\widehat{=}$

Utility event. Marks a single page specified by the given row address as obsolete.

extends *mark_a_page_obsolete*

where

grd3 : $old\_r \notin dom(relocating\_pages)$

end

Event *erase_a_block* $\widehat{=}$

Erase all the given pages within the given block, which is obsolete.

extends *erase_a_block*

Event *start_relacating_a_block* $\widehat{=}$

(1) Start relocating a block b (which is a candidate) if the given block has been marked as obsolete. The relocating block becomes obsolete when there are no valid pages.

any

b

where

grd1 : $b \in BLOCK$

grd2 : $b \notin relocating\_blocks$

grd3 : $\forall r \cdot r \in RowAddr \land BlkOfRow(r) = b \Rightarrow r \notin ran(corresRowOfwritingPR) \cup ran(corresRowOfreadingPR)$

There is no page being written or read.

grd4 : $\exists r \cdot r \in (programmed\_pages \cup programmed\_pages2) \land BlkOfRow(r) = b \Rightarrow r \notin obsolete\_pages2$

At least one valid page exists

then

act1 : $relocating\_blocks := relocating\_blocks \cup \{b\}$

end

Event *start_relocating_a_page* $\widehat{=}$

(2.1) Start relocating a valid page within the relocating block (b), if exist, from *old_r* to *new_r*

any

old_r, new_r, b

where

grd1 : $old\_r \in (programmed\_pages \cup programmed\_pages2) \setminus obsolete\_pages2$

grd2 : $b \in relocating\_blocks$

grd3 : $BlkOfRow(old\_r) = b$

grd4 : $new\_r \in RowAddr \setminus (programmed\_pages \cup programmed\_pages2)$

grd5 : $old\_r \notin dom(relocating\_pages)$

grd6 : $new\_r \notin ran(relocating\_pages)$

then

act1 : $relocating\_pages := relocating\_pages \cup \{old\_r \mapsto new\_r\}$

end

Event *copy_a_page_to_new_loc* $\widehat{=}$

(2.2) Write the content of page at the old location to another location.

refines *copy_a_page_to_new_loc*

any

old_r, new_r

$pdata$ a PDATA to be copied from *old_r* to *new_r*

where

grd1 : $old\_r \mapsto new\_r \in relocating\_pages$

grd2 : $new\_r \notin programmed\_pages2$

grd4 : $pdata = flash(old\_r)$

grd5 : $new\_r \notin ran(trans\_func)$

then
act1 : $flash2(new\_r) := pdata$
act2 : $programmed\_pages2 := programmed\_pages2 \cup \{new\_r\}$
act3 : $trans\_func(old\_r) := new\_r$
end

Event *mark_old_page_obsolete* $\widehat{=}$

(2.3a) Mark the old page to be obsolete at the end when the content has been written to the new location.

refines *mark_old_page_obsolete*

any
old_r, new_r
where
grd1 : $old\_r \mapsto new\_r \in relocating\_pages$
grd2 : $new\_r \in programmed\_pages2$
then
act1 : $obsolete\_pages2 := obsolete\_pages2 \cup \{old\_r\}$
act2 : $relocating\_pages := relocating\_pages \setminus \{old\_r \mapsto new\_r\}$
end

Event *relocate_a_page_fail* $\widehat{=}$

(2.3b) In the case of relocating the given page fails (or abort), remove the tuple of pages being located. If locating a page is aborted at any point, (i) fail to write to a new location (fail at 2.2), the content at the old location is still valid; (ii) fail to mark the old as obsolete. That means there two valid pages with the same content in both old and new location. However, when the flash is remounted only one is selected to formulate the fat table and then mark another obsolete.

any
old_r, new_r
where
grd1 : $old\_r \mapsto new\_r \in relocating\_pages$
then
act1 : $relocating\_pages := relocating\_pages \setminus \{old\_r \mapsto new\_r\}$
end

Event *relocate_a_block_end* $\widehat{=}$

(3a success) Mark the block being located as obsolete when there are no valid pages exist. The obsolete block is the block that is read for erasing.

any
b
where
grd1 : $b \in relocating\_blocks$
grd2 : $\forall r \cdot r \in (programmed\_pages2 \cup progrmmed\_pages) \wedge BlkOfRow(r) = b \Rightarrow r \in obsolete\_pages2)$
No valid pages within the given block.
then
act1 : $relocating\_blocks := relocating\_blocks \setminus \{b\}$
end

Event *relocate_a_block_fail* $\widehat{=}$

(3b fail) When relocating a block fails. As the result, some valid pages may exist and it has not been marked as obsolete. That means this block might be selected to relocate and erase again in the future.

any

b the block being relocated
rws all rows within the given block

where
grd1 : $b \in relocating\_blocks$
grd2 : $rws = BlkOfRow^{-1}[\{b\}]$

then
act1 : $relocating\_blocks := relocating\_blocks \setminus \{b\}$
act2 : $relocating\_pages := rws \lhd relocating\_pages$

end

END

C.5 The fourth refinement: Refining the *block_erase* event

MACHINE FMCH07_FL_REF4
REFINES FMCH07_FL_REF3
SEES FLCTX3
VARIABLES

`flash`
`programmed_pages`
`obsolete_pages`
`ready2read` page registers that their data are ready to be read
`ready2prog` page registers that their data are ready to be programmed into the flash
`readingPR` Set of page registers being in the reading state.
`writingPR` Set of page registers being in the writing state.
`corresRowOfreadingPR`
`corresRowOfwritingPR`
`dataOfPR` data of each page register
`flash2` represents part of flash that have been programmed during relocation
`trans_func` A translation function, mapping the content from the old location to the new location.
`programmed_pages2` represents a set of pages that have already been programmed during the relocating process
`obsolete_pages2` represents a set of all obsolete pages
`relocating_blocks` blocks in the relocating state
`relocating_pages` pairs of pages (old,new) that are in the relocating state
candidate_blocks blocks which are candidate to be relocated
obsolete_blocks set (programmed) blocks that have no valid pages
erasing_blocks blocks being in the erasing state
num_erased the number of times that each block has been erased
invalid_num_erased_blocks (erased) blocks with invalid num_erased
restoring_num_erased blocks in the restoring num_erased state
tmp_num_erased temporary places storing the number of erasures
corresBlkOftmpErased the corresponding block of the tmp_num_erased
bad_blocks set of bad blocks

INVARIANTS

inv1 : $candidate\_blocks \subseteq BLOCK$
inv2 : $relocating\_blocks \subseteq candidate\_blocks$
inv3 : $obsolete\_blocks \subseteq BLOCK$
inv4 : $obsolete\_blocks \cap relocating\_blocks = \varnothing$
inv5 : $\forall r \cdot r \in (programmed\_pages \cup programmed\_pages2) \land BlkOfRow(r) \in obsolete\_blocks \Rightarrow r \in obsolete\_pages2$
inv6 : $erasing\_blocks \subseteq obsolete\_blocks$
inv7 : $\forall b, r \cdot b \in obsolete\_blocks \land r \in RowAddr \land BlkOfRow(r) = b \Rightarrow r \notin ran(corresRowOfreadingPR) \land r \notin ran(corresRowOfwritingPR)$

inv8 : $num\_erased \in BLOCK \rightarrow \mathbb{N}$
inv9 : $invalid\_num\_erased\_blocks \subseteq BLOCK$
inv10 : $restoring\_num\_erased \subseteq invalid\_num\_erased\_blocks$
inv11 : $tmp\_num\_erased \in RowAddr \nrightarrow \mathbb{N}$
inv12 : $corresBlkOftmpErased \in dom(tmp\_num\_erased) \rightarrow BLOCK$
inv13 : $bad\_blocks \subseteq BLOCK$
inv14 : $bad\_blocks \cap candidate\_blocks = \varnothing$
inv15 : $\forall r \cdot r \in dom(relocating\_pages) \Rightarrow BlkOfRow(r) \in relocating\_blocks$

EVENTS

Initialisation

extended

begin
act19 : $candidate\_blocks := \varnothing$
act20 : $obsolete\_blocks := \varnothing$
act21 : $erasing\_blocks := \varnothing$
act22 : $num\_erased := BLOCK \times \{0\}$
act27 : $invalid\_num\_erased\_blocks := \varnothing$
act23 : $restoring\_num\_erased := \varnothing$
act24 : $tmp\_num\_erased := \varnothing$
act25 : $corresBlkOftmpErased := \varnothing$
act26 : $bad\_blocks := \varnothing$
end

Event *pageread_start* $\widehat{=}$

(r1) Starting page_read by selecting the related LUN (in which the the page row r is) and avialable page register.

extends *pageread_start*

where
grd7 : $BlkOfRow(r) \notin obsolete\_blocks \cup relocating\_blocks \cup bad\_blocks$
end

Event *read2reg* $\widehat{=}$

(r2) Transfer the page-data at the given row address to the page register. (case1)

extends *read2reg*

Event *read2reg2* $\widehat{=}$

(r2) Transfer the page-data from the given row address to the page register. (case2)

extends *read2reg2*

Event *page_read_end* $\widehat{=}$

(r3.ok) Read page data from the register off chip.

extends *page_read_end*

Event *page_read_fail* $\widehat{=}$

(r3.fail)

extends *page_read_fail*

Event *pageprog_start* $\widehat{=}$

(w1) Starting page_programmed by selecting the related LUN and avialable page register.

extends *pageprog_start*

where
grd10 : $BlkOfRow(r) \notin obsolete\_blocks \cup relocating\_blocks \cup bad\_blocks$
end

Event *write2reg* $\widehat{=}$

(w2) Write/transfer the data to be programmed into the page register within the LUN.

extends *write2reg*

Event *page_program_ok* $\widehat{=}$

(w3.ok) Programme the data on the page register to the flash at the corresponding row address.

extends *page_program_ok*

Event *page_prog_fail* $\widehat{=}$

(w3.fail) programming the given page fails

extends *page_prog_fail*

Event *mark_pages_obsolete* $\widehat{=}$

A utility event used for marking a set of pages as obsolete.

extends *mark_pages_obsolete*

Event *mark_a_page_obsolete* $\widehat{=}$

Utility event. Marks a single page specified by the given row address as obsolete.

extends *mark_a_page_obsolete*

Event *start_relacating_a_block* $\widehat{=}$

(1) Start relocating a block (which is a candidate) if the given block has been marked as obsolete. The relocating block becomes obsolete when there are no valid pages.

refines *start_relacating_a_block*

any
b
where
grd1 : $b \in candidate\_blocks$
grd2 : $b \notin relocating\_blocks \cup obsolete\_blocks$
grd3 : $\forall r \cdot r \in RowAddr \wedge BlkOfRow(r) = b \Rightarrow r \notin ran(corresRowOfwritingPR) \cup ran(corresRowOfreadingPR)$
There is no page being written or read.
grd4 : $BlkOfRow^{-1}[\{b\}] \cap (programmed\_pages \setminus obsolete\_pages2) \neq \varnothing$
Existing some valid pages
then
act1 : $relocating\_blocks := relocating\_blocks \cup \{b\}$
end

Event *start_relocating_a_page* $\widehat{=}$

(2.1) Start relocating a valid page within the relocating block (if exist).

extends *start_relocating_a_page*

Event *copy_a_page_to_new_loc* $\widehat{=}$

(2.2) Write the content of page at the old location to another location.

extends *copy_a_page_to_new_loc*

where

grd6 : $BlkOfRow(new\_r) \notin obsolete\_blocks \cup bad\_blocks$

end

Event *mark_old_page_obsolete* $\widehat{=}$

(2.3) Mark the old page to be obsolete at the end when the content has been written to the new location.

extends *mark_old_page_obsolete*

Event *relocate_a_page_fail* $\widehat{=}$

(2.fail) In the case of relocating the given page fails (or abort), remove the tuple of pages being located. If locating a page is aborted at any point, (2.2.fail) fail to write to a new location (fail at 2.2), the content at the old location is still valid; (2.3.fail) fail to mark the old as obsolete. That means there are two valid pages with the same content in both old and new location. However, when the flash is mounted only one is selected to formulate the fat table and then mark another obsolete.

extends *relocate_a_page_fail*

Event *relocate_a_block_end* $\widehat{=}$

(3.ok) Mark the block being located as obsolete when there are no valid pages exist. The obsolete block is the block that is read for erasing.

refines *relocate_a_block_end*

any

b

where

grd1 : $b \in relocating\_blocks$

or $lid \mapsto bid \in LUAddr \times BAddr$ meaning it can be any block. The dirty block is a candidate block to be reclaim.

grd2 : $BlkOfRow^{-1}[\{b\}] \cap (programmed\_pages \setminus obsolete\_pages2) = \varnothing$

No valid pages within the given block.

then

act1 : $r\,relocating\_blocks := relocating\_blocks \setminus \{b\}$

act2 : $obsolete\_blocks := obsolete\_blocks \cup \{b\}$

end

Event *relocate_a_block_fail* $\widehat{=}$

(3.fail) When relocating a block fails. As the result, some valid pages may exist and it has not been marked as obsolete. That means this block might be selected to relocate and erase again in the future.

refines *relocate_a_block_fail*

any

b, rws

where

grd1 : $b \in relocating\_blocks$

grd2 : $rws = BlkOfRow^{-1}[\{b\}]$

then

act1 : $r\,relocating\_blocks := relocating\_blocks \setminus \{b\}$

act2 : $r\,relocating\_pages := rws \lhd relocating\_pages$

end

Event *start_erase_block* $\widehat{=}$

(4.1) Start erasing an obsolete block. Set the given block in the erasing state.

[Store the number of erasures somewhere else (in the other block) before erasing. The number of erasure will be restored when erasing process complete]
This step are not necessary to be performed once the relocation of the given block complete. This event just pick up one of the obsolete blocks to be erased.

any
b
$free\_r$
where
grd1 : $b \in obsolete\_blocks$
grd2 : $b \notin erasing\_blocks$
grd3 : $num\_erased(b) \leq max\_erase$
grd4 : $free\_r \in RowAddr \setminus (programmed\_pages2 \cup programmed\_pages)$
grd5 : $BlkOfRow(free\_r) \notin erasing\_blocks$
grd6 : $free\_r \notin dom(tmp\_num\_erased)$
grd7 : $\forall k \cdot k \in obsolete\_blocks \setminus bad\_blocks \Rightarrow num\_eraseOfblock(k) \geq num\_eraseOfblock(b)$

select an obsolete block with the least number of erasures
grd8 : $b \notin bad\_blocks$
then
act1 : $erasing\_blocks := erasing\_blocks \cup \{b\}$
act2 : $tmp\_num\_erased(free\_r) := num\_erased(b)$
Store the number of erasures somewhere else.
act3 : $corresBlkOftmpErased(free\_r) := b$
end

Event *erase_a_block_end* $\widehat{=}$
(4.2.ok) erase the given block which is obsolete. That means all pages have already set to the default (dp). The previous num erased is also cleared. That is the number of erasing times is now invalid.

refines *erase_a_block*

any
$rows$ All rows within the given block.
b
where
grd1 : $b \in erasing\_blocks$
grd2 : $rows = BlkOfRow^{-1}[\{b\}]$
grd3 : $rows \cap ((programmed\_pages2 \cup programmed\_pages) \setminus obsolete\_pages) = \varnothing$
The block being erased have no valid pages
grd4 : $rows \cap ran(corresRowOfreadingPR) = \varnothing$
grd5 : $rows \cap ran(corresRowOfwritingPR) = \varnothing$
grd6 : $rows \cap dom(trans\_func) = \varnothing$
$\forall r \cdot r \in dom(trans\_func) \Rightarrow flash(r) = dp$
grd7 : $rows \cap ran(trans\_func) = \varnothing$
then
act1 : $flash := flash \lhd (rows \times \{dp\})$
act2 : $programmed\_pages := programmed\_pages \setminus rows$
act3 : $obsolete\_pages := obsolete\_pages \setminus rows$
act4 : $programmed\_pages2 := programmed\_pages2 \setminus rows$
act5 : $obsolete\_pages2 := obsolete\_pages2 \setminus rows$
act6 : $erasing\_blocks := erasing\_blocks \setminus \{b\}$
act7 : $obsolete\_blocks := obsolete\_blocks \setminus \{b\}$

act8 : $candidate\_blocks := candidate\_blocks \setminus \{b\}$
act9 : $invalid\_num\_erased\_blocks := invalid\_num\_erased\_blocks \cup \{b\}$
end

Event *erase_a_block_fail* $\widehat{=}$

(4.2.fail) In the case of erasing fails. The block is still in the obsolete state that might be selected to be erased later. (The obsolete block is invalid to be used)

any
b
where
grd1 : $b \in erasing\_blocks$
then
act1 : $erasing\_blocks := erasing\_blocks \setminus \{b\}$
end

Event *start_restore_num_erased* $\widehat{=}$

(5.1) Start restoring the number of erasures

any
b
where
grd1 : $b \in invalid\_num\_erased\_blocks$
then
act1 : $restoring\_num\_erased := restoring\_num\_erased \cup \{b\}$
end

Event *restore_num_erased* $\widehat{=}$

(5.2.ok) Restoring of the number of erasures success.

any
b
row the row that temporarily stores the number of times that block b has been erased
where
grd1 : $b \in restoring\_num\_erased$
grd2 : $row \in dom(tmp\_num\_erased)$
grd3 : $b = corresBlkOftmpErased(row)$
then
act1 : $num\_erased(b) := tmp\_num\_erased(row) + 1$
act2 : $restoring\_num\_erased := restoring\_num\_erased \setminus \{b\}$
act3 : $tmp\_num\_erased := \{row\} \lhd tmp\_num\_erased$
act4 : $corresBlkOftmpErased := \{row\} \lhd corresBlkOftmpErased$
act5 : $invalid\_num\_erased\_blocks := invalid\_num\_erased\_blocks \setminus \{b\}$
end

Event *restore_num_erased_fail* $\widehat{=}$

(5.2.fail) Restoring of the number of erasures fails. This means the num_erased of this block still invalid. It may be restored later, since the valid one still remain.

any
b
where
grd1 : $b \in restoring\_num\_erased$
then
act2 : $restoring\_num\_erased := restoring\_num\_erased \setminus \{b\}$
end

END

C.6 The fifth refinement: Status Register

MACHINE FMCH07_FL_REF5
REFINES FMCH07_FL_REF4
SEES FLCTX4
VARIABLES

`flash`
`programmed_pages`
`obsolete_pages`
`ready2read` page registers that their data are ready to be read
`ready2prog` page registers that their data are ready to be programmed into the flash
`readingPR` Set of page registers being in the reading state.
`writingPR` Set of page registers being in the writing state.
`corresRowOfreadingPR`
`corresRowOfwritingPR`
`dataOfPR` data of each page register
`flash2` represents part of flash that have been programmed during relocation
`trans_func` A translation function, mapping the content from the old location to the new location.
`programmed_pages2` represents a set of pages that have already been programmed during the relocating process
`obsolete_pages2` represents a set of all obsolete pages
`relocating_blocks` blocks in the relocating state
`relocating_pages` pairs of pages (old,new) that are in the relocating state
`candidate_blocks` blocks which are candidate to be relocated
`obsolete_blocks` set (programmed) blocks that have no valid pages
`erasing_blocks` blocks being in the erasing state
`num_erased` the number of times that each block has been erased
`invalid_num_erased_blocks` (erased) blocks with invalid num_erased
`restoring_num_erased` blocks in the restoring num_erased state
`tmp_num_erased` temporary places storing the number of erasures
`corresBlkOftmpErased` the corresponding block of the tmp_num_erased
`bad_blocks` set of bad blocks
t_status the status of the target flash device
$lready$ set of LUNs that their status values is ready
$lnotready$ set of LUNs that their status values is not ready
$lreadyfail$ set of LUNs that is ready, but the previous command fails
$wprotected$ set of LUNs that are write protected

INVARIANTS

inv1 : $t\_status \in STATUS$
inv2 : $lready \subseteq LUAddr$
inv3 : $lnotready \subseteq LUAddr$
inv4 : $lreadyfail \subseteq LUAddr$
inv5 : $wprotected \subseteq LUAddr$
inv6 : $partition(LUAddr, lready, lnotready, lreadyfail)$

inv7 : $t\_status = RDY \Rightarrow (\forall\, l \cdot l \in LUAddr \Rightarrow l \in (lready \cup lreadyfail))$
inv8 : $\forall r \cdot r \in RowAddr \wedge lidOfRow(r) \in wprotected \Rightarrow r \notin ran(relocating\_pages)$
inv9 : $\forall r \cdot r \in RowAddr \wedge r \in ran(relocating\_pages) \Rightarrow lidOfRow(r) \notin wprotected$

EVENTS

Initialisation

extended

begin
act27 : $t\_status := RDY$
act28 : $lready := LUAddr$
act29 : $lnotready := \varnothing$
act30 : $lreadyfail := \varnothing$
act31 : $wprotected := \varnothing$
end

Event *pageread_start* $\widehat{=}$

(r1) Starting page_read by selecting the related LUN (in which the the page no. r is) and available page register.

extends *pageread_start*

where
grd12 : $t\_status = RDY$
grd11 : $lid \notin wprotected$
then
act3 : $lnotready := lnotready \cup \{lid\}$
act4 : $lready := lready \setminus \{lid\}$
act5 : $lreadyfail := lreadyfail \setminus \{lid\}$
act6 : $t\_status := nRDY$
end

Event *read2reg* $\widehat{=}$

(r2) Transfer the page-data at the given row address to the page register.

extends *read2reg*

Event *page_read_end* $\widehat{=}$

(r3a) Read page data from the register off chip.

extends *page_read_end*

any
lid
where
grd5 : $lid = lidOfRow(r)$
grd6 : $lid \in lnotready$
then
act5 : $lready := lready \cup \{lid\}$
act6 : $lnotready := lnotready \setminus \{lid\}$
end

Event *page_read_fail* $\widehat{=}$

(r3b)

extends *page_read_fail*

any
lid
where
grd3 : $lid = lidOfRow(r)$
grd4 : $lid \in lnotready$
then

act4 : $lreadyfail := lreadyfail \cup \{lid\}$
act5 : $lnotready := lnotready \setminus \{lid\}$
end

Event *pageprog_start* $\widehat{=}$

(w1) Starting page_programmed by selecting the related LUN and avialable page register.

extends *pageprog_start*

where
grd12 : $t\_status = RDY$
grd13 : $lid \notin wprotected$
then
act3 : $lnotready := lnotready \cup \{lid\}$
act4 : $lready := lready \setminus \{lid\}$
act5 : $lreadyfail := lreadyfail \setminus \{lid\}$
act6 : $t\_status := nRDY$
end

Event *write2reg* $\widehat{=}$

(w2) Write/transfer the data to be programmed into the page register within the LUN.

extends *write2reg*

Event *page_program_ok* $\widehat{=}$

(w3a) Programme the data on the page registier to the flash at the corresponding row address.

extends *page_program_ok*

any
lid
where
grd8 : $lid \in lnotready$
grd9 : $lid = lidOfRow(new\_r)$
then
act8 : $lready := lready \cup \{lid\}$
act9 : $lnotready := lnotready \setminus \{lid\}$
end

Event *page_prog_fail* $\widehat{=}$

(w3b)

extends *page_prog_fail*

any
lid
where
grd3 : $lid \in lnotready$
grd4 : $lid = lidOfRow(r)$
then
act5 : $lreadyfail := lreadyfail \cup \{lid\}$
act6 : $lnotready := lnotready \setminus \{lid\}$
end

Event *mark_pages_obsolete* $\widehat{=}$

A utility event used for marking a set of pages as obsolete.

extends *mark_pages_obsolete*

Event *mark_a_page_obsolete* $\widehat{=}$

Utility event. Marks a single page specified by the given row address as obsolete.

extends *mark_a_page_obsolete*

Event *start_relacating_a_block* $\widehat{=}$

(1) Start relocating a block (which is a candidate) if the given block has been marked as obsolete. The relocating block becomes obsolete when there are no valid pages.

extends *start_relacating_a_block*

Event *start_relocating_a_page* $\widehat{=}$

(2.1) Start relocating a valid page within the relocating block (if exist).

extends *start_relocating_a_page*

any
lid_old, lid_new
where
grd9 : $lid\_old = lidOfRow(old\_r)$
grd10 : $lid\_new = lidOfRow(new\_r)$
grd11 : $\{lid\_new, lid\_old\} \cap wprotected = \varnothing$
then
act2 : $lnotready := lnotready \cup \{lid\_old, lid\_new\}$
act3 : $lready := lready \setminus \{lid\_old, lid\_new\}$
act4 : $lreadyfail := lreadyfail \setminus \{lid\_old, lid\_new\}$
act5 : $t\_status := nRDY$
end

Event *copy_a_page_to_new_loc* $\widehat{=}$

(2.2) Write the content of page at the old location to another location.

extends *copy_a_page_to_new_loc*

Event *mark_old_page_obsolete* $\widehat{=}$

(2.3) Mark the old page to be obselete at the end when the content has been written to the new location.

extends *mark_old_page_obsolete*

Event *relocate_a_page_fail_case1* $\widehat{=}$

(2.2.fail) Fail to write to a new location (fail at 2.2), the content at the old location is still valid

extends *relocate_a_page_fail*

any
lid
where
grd2 : $new\_r \notin programmed\_pages2$
grd3 : $lid = lidOfRow(new\_r)$
grd4 : $lid \in lnotready$
then
act2 : $lreadyfail := lreadyfail \cup \{lid\}$
act3 : $lnotready := lnotready \setminus \{lid\}$
end

Event *relocate_a_page_fail_case2* $\widehat{=}$

(2.3.fail) Fail to mark the old as obsolete. That means there are two valid pages with the same content in both old and new location. However, when the flash is mounted only one is selected to formulate the fat table and then mark another obsolete.

extends *relocate_a_page_fail*

any
lid
where
grd2 : $new\_r \in programmed\_pages2$
grd3 : $lid = lidOfRow(old\_r)$
grd4 : $lid \in lnotready$
grd5 : $old\_r \notin obsolete\_pages2$
then
act2 : $lreadyfail := lreadyfail \cup \{lid\}$
act3 : $lnotready := lnotready \setminus \{lid\}$
end

Event *relocate_a_block_end* $\widehat{=}$

(3.ok) Mark the block being located as obsolete when there are no valid pages exist. The obsolete block is the block that is read for erasing.

extends *relocate_a_block_end*

any
lid
where
grd4 : $lid \in lnotready$
grd5 : $lid = lidOfBlk(b)$
then
act3 : $lready := lready \cup \{lid\}$
act4 : $lnotready := lnotready \setminus \{lid\}$
end

Event *relocate_a_block_fail* $\widehat{=}$

(3.fail) When relocating a block fails. As the result, some valid pages may exist and it has not been marked as obsolete. That means this block might be selected to relocate and erase again in the future.

extends *relocate_a_block_fail*

any
lid
where
grd3 : $lid \in lnotready$
grd4 : $lid = lidOfBlk(b)$
then
act3 : $lreadyfail := lreadyfail \cup \{lid\}$
act4 : $lnotready := lnotready \setminus \{lid\}$
end

Event *start_erase_block* $\widehat{=}$

(4.1) Start erasing an obsolete block. Set the given block in the erasing state. [Store the number of erasures somewhere else (in the other block) before erasing. The number of erasure will be restored when erasing process complete]

extends *start_erase_block*

any
lid
where
grd9 : $t\_status = RDY$
grd10 : $lid = lidOfBlk(b)$
grd11 : $lid \notin wprotected$
then
act4 : $lnotready := lnotready \cup \{lid\}$
act5 : $lready := lready \setminus \{lid\}$

act6 : $lreadyfail := lreadyfail \setminus \{lid\}$
act7 : $t\_status := nRDY$
end

Event *erase_a_block_ok* $\widehat{=}$

(4.2.ok) erase the given block which is obsolete

extends *erase_a_block*

any
lid
where
grd8 : $lid \in lnotready$
grd10 : $lid = lidOfBlk(b)$
then
act8 : $lready := lready \cup \{lid\}$
act9 : $lnotready := lnotready \setminus \{lid\}$
end

Event *erase_a_block_fail* $\widehat{=}$

(4.2.fail) In the case of erasing fails. The block is still in the obsolete state that might be selected to be erased later. (The obsolete block is invalid to be used.)

extends *erase_a_block_fail*

any
lid
where
grd2 : $lid \in lnotready$
grd3 : $lid = lidOfBlk(b)$
then
act5 : $lreadyfail := lreadyfail \cup \{lid\}$
act6 : $lnotready := lnotready \setminus \{lid\}$
end

Event *start_restore_num_erased* $\widehat{=}$

(5.1) Start restoring the number of erasures

extends *start_restore_num_erased*

any
lid
where
grd9 : $t\_status = RDY$
grd10 : $lid = lidOfBlk(b)$
grd11 : $lid \notin wprotected$
then
act4 : $lnotready := lnotready \cup \{lid\}$
act5 : $lready := lready \setminus \{lid\}$
act6 : $lreadyfail := lreadyfail \setminus \{lid\}$
act7 : $t\_status := nRDY$
end

Event *restore_num_erased_ok* $\widehat{=}$

(5.2.ok) restore the number of erasures at the end of erasing a block

extends *restore_num_erased*

any
lid
where
grd5 : $lid = lidOfBlk(b)$
grd4 : $lid \in lnotready$
then
act8 : $lready := lready \cup \{lid\}$

act9 : $lnotready := lnotready \setminus \{lid\}$

end

Event *restore_num_erased_fail* $\widehat{=}$

(5.2.fail) Restoring of the number of erasures fails. This means the num_erased of this block still invalid. It may be restored later, since the the vilid one still remian.

extends *restore_num_erased_fail*

any

lid

where

grd2 : $lid = lidOfBlk(b)$

grd3 : $lid \in lnotready$

then

act2 : $lready := lready \cup \{lid\}$

act3 : $lnotready := lnotready \setminus \{lid\}$

end

Event *setwprotect* $\widehat{=}$

any

lid

where

grd1 : $lid \in LUAddr$

grd2 : $lid \notin wprotected$

grd3 : $t\_status = RDY$

grd4 : $ran(relocating\_pages) \cap lidOfRow^{-1}[\{lid\}] = \varnothing$

No pages being in the relocating process.

grd5 : $ran(corresRowOfwritingPR) \cap lidOfRow^{-1}[\{lid\}] = \varnothing$

No page being programmed

then

act1 : $wprotected := wprotected \cup \{lid\}$

end

Event *remove_wprotect* $\widehat{=}$

any

lid

where

grd1 : $lid \in LUAddr$

grd2 : $lid \in wprotected$

grd3 : $t\_status = RDY$

then

act1 : $wprotected := wprotected \setminus \{lid\}$

end

Event *read_flash_status* $\widehat{=}$

any

st

where

grd1 : $st \in \{RDY, nRDY\}$

grd2 : $st = RDY \Leftrightarrow (\forall l \cdot l \in LUAddr \Rightarrow l \in (lready \cup lreadyfail))$

then

act1 : $t\_status := st$

end

END

Bibliography

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial. A system development process with Event-B and the Rodin platform. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2007.

[3] J.-R. Abrial. Formal methods: Theory becoming practice. *J. UCS*, 13(5):619–628, May 2007.

[4] J.-R. Abrial. *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[5] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 10.1007, April 2010. Published online.

[6] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006 Lecture Notes in Computer Science*, volume 4260, 2006.

[7] J-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. A roadmap for the Rodin toolset version 1.0: 12 June 2008. In Börger et al. [19], page 347.

[8] J.-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Infomatica*, 2006.

[9] S. Agerholm and P. G. Larsen. The IFAD VDM tools: Lightweight formal methods. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 326–329, London, UK, 1999. Springer-Verlag.

[10] R.-J. Back, , and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[11] R. J. R. Back. Refinement calculus, part ii: parallel and reactive programs. In *REX workshop: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 67–93, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[12] R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10:513–554, 1988.

[13] J.C.M. Baeten. A brief history of process algebra. Technical report, Department of Computer Science, Technische Universiteit Eindhoven, 2004.

[14] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Programmiermethodik. Formal system development with kiv. In *Fundamental Approaches to Software Engineering, number 1783 in LNCS*, pages 363–366. Springer, 2000.

[15] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 100–111, London, UK, 2006. Springer-Verlag.

[16] J. Bendisposto and M. Leuschel. The ProB plug-in for Eclipse and Rodin. Technical report, Institut für Informatik, Heinrich-Heine Universität Düsseldorf, 2007.

[17] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[18] E. Börger. The abstract state machines method for high-level system design and analysis. Technical report, Dipartimento di Informatica, Universita di Pisa, Italy, 2003.

[19] E. Börger, M. Butler, J. P. Bowen, and Paul Boca, editors. *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*. Springer, 2008.

[20] J. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 2003.

[21] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[22] M. Butler. *A CSP approach to action systems*. PhD thesis, Programming Research Group, Oxford University, 1992.

[23] M. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.

[24] M. Butler. On the verified-by-construction approach. Technical report, University of Southampton, UK, February 2006.

[25] M. Butler. Rodin deliverable D31: Plublic versions of plug-in tools. Technical report, University of Southampton, UK, 2007. Available from: http://rodin.cs.ncl.ac.uk/D31.pdf.

[26] M. Butler. Decomposition structures for Event-B. In *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 20–38, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] M. Butler, J.-R. Abrial, K. Damchoom, and A. Edmunds. Applying Event-B and Rodin to the filestore, 2008. VSRNet Workshop, ABZ 2008.

[28] M. Butler, J. Grundy, T. Langbacka, R. Ruksenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In *Formal Methods Pacific 97*, pages 40–61. Springer, 1997.

[29] M. Butler and S. Hallerstede. The Rodin formal modelling tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.

[30] M. Butler, M. Leuschel, and C. Snook. Tools for system validation with B abstract machines. In *ASM 2005: 12th International Workshop on Abstract State Machines*, 2005.

[31] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2007.

[32] R. W. Butler. What is formal methods. Technical report, NASA, 2001. Available from http://shemesh.larc.nasa.gov/fm/fm-what.html.

[33] A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[34] A. Butterfield and J. Woodcock. Formalising flash memory: First steps. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 251–260, Washington, DC, USA, 2007. IEEE Computer Society.

[35] D. Cansell and D. Méry. Foundation of the B-method. *Computing and Informatics*, 20:1–31, 2003.

[36] D. Cansell and D. Méry. Tutorial on the event-based B method Concepts and Case Studies. Technical report, LORIA and Université Henri Poincaré Nancy, 2006.

[37] D. Carrington. Vdm and the refinement calculus: a comparison of two systematic design methods. Technical report, The University of Queenland, Australia, December 1993.

[38] ClearSy. Atelier B translators user manual, version 4.6. Technical report, ClearSy, Parc de la Duranne, 2009. Available from http://www.atelierb.eu/ressources/DOC/english/translators-user-manual.pdf.

[39] T. Clement. Combining transformation and posit-and prove in a VDM development. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 63–80, London, UK, 1991. Springer-Verlag.

[40] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. Rodin (rigorous open development environment for complex systems). *Fifth European Dependable Computing Conference: EDCC-5 supplementary*, pages 23–26, 2005. Available from http://rodin.cs.ncl.ac.uk.

[41] CSK System Corp. VDMTools, 2009. Available from http://www.vdmtools.jp.

[42] Oracle Corporation. Object-oriented programming concepts, Jan 2010. http://java.sun.com/docs/books/tutorial/java/concepts/.

[43] K. Damchoom and M. Butler. An experiment in applying Event-B and Rodin to a flash-based filestore. In *Rodin User and Developer Workshop*, July 2009.

[44] K. Damchoom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, volume 5902 of *Lecture Notes in Computer Science*, pages 134–152. Springer, 2009.

[45] K. Damchoom, M. J. Butler, and J.-R. Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 25–44. Springer, 2008.

[46] K. Damchoom and E. R. Jam. B2Latex, a LaTeX code generator for the Rodin platform, 2007. Available from: http://www.event-b.org.

[47] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[48] A. Edmunds and M. Butler. Linking Event-B and concurrent object-oriented programs. *Electronic Notes in Theoretical Computer Science*, 2008.

[49] A. Edmunds and M. Butler. Tool support for Event-B code generation. In *Workshop on Tool Building in Formal Methods - ABZ Conference, Canada*, 2010.

[50] S. Eisenbach and R. Paterson. $\pi-$calculus semantics for the concurrent configuration language darwin. In *Hawaii International Conference on System Sciences*, Koloa, Hawaii, January 1993.

[51] Hynix Semiconductor et al. Open NAND flash interface specification, revision 1.0. Technical report, ONFI, www.onfi.org, Dec. 2006.

[52] Hynix Semiconductor et al. Open nand flash interface specification, revision 2.0. Technical report, ONFI, www.onfi.org, 2008.

[53] N. Evans and M. Butler. A proposal for records in event-b. In Tobias Nipkow, Jayadev Misra, and Emil Sekerinski, editors, *Formal Methods 2006*, volume LNCS 4085, pages 221–235. Springer, 2006.

[54] M. A. Ferreira, S. S. Silva, and J. N. Oliveira. Verifying intel flash file system core specification. In *Fourth VDM/Overture Workshop (CS-TR-1099)*, 2008.

[55] R. B. Findler. Scheme and functional programming 2006: paper abstracts. *SIGPLAN Not.*, 41(8):6–9, 2006.

[56] D. Flanagan. *Java in a Nutshell*. O'Reilly, USA, 5th edition, 2005.

[57] The Eclipse Foundation. Eclipse - an open development platform, 2007. Available from http://www.eclipse.org/.

[58] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In *ICECCS*, pages 3–14. IEEE Computer Society, 2007.

[59] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. *ICECCS*, 0:153–162, 2008.

[60] L. Freitas, J. Woodcock, and Z. Fu. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. Comput. Program.*, 74(4):238–257, 2009.

[61] Z. Fu. *A refinement of the Unix filling system using Z/Eves*. PhD thesis, University of York, 2006.

[62] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[63] B. Goetz. *Java Concurrency in Practice*. Addison Wesley, USA, 2006.

[64] M. Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[65] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[66] The Open Group. IEEE Std 1003.1, 2004 Edition. Available from http://www.unix.org/version3/ieee_std.html.

[67] Intel Flash File System Core Reference Guide. Vesion 1. Technical Report 304436-001, Intel Corporation, Oct 2004.

[68] S. Hallerstede. Justification for the Event-B modelling notation. *LNCS*, 4255:49–63, 2007.

[69] R. Harper. *Programming in Standard ML.* Carnegie Mellon University, 2005. Available from: http://www.cs.cmu.edu/ rwh/smlbook/online.pdf.

[70] W. H. Hesselink and M. I. Lali. Formalizing a hierarchical file system. *Electron. Notes Theor. Comput. Sci.*, 259:67–85, 2009.

[71] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 2004.

[72] T. Hoare and J. Misra. Verified software: theories, tools, experiments; vision of a grand challenge project. 2005.

[73] C. M. Holloway. Why engineers should consider formal methods. In *The 16th Digital Avionics Systems Conference*, 1997.

[74] G. J. Holzmann. Promela language reference. Available from http://www.spinroot.com/spin/Man/promela.html.

[75] G. J. Holzmann. *The Spin model checker: primer and reference manual.* Addison Wesley, USA, 2004.

[76] G. J. Holzmann, R. Joshi, and A. Groce. New challenges in model checking. In *Proceedings of the Conference on Computer-Aided Verification.* Pasadena, CA : Jet Propulsion Laboratory, NASA, 2006. Available from http://hdl.handle.net/2014/39859.

[77] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[78] J. Hughes. Specifying a visual file system in Z. Technical report, Department of Computing Science, University of Glasgow, 1989.
Available from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=199162.

[79] G. Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[80] A. Ireland, G. Grov, and M. Butler. Reasoned modelling critics: turning failed proofs into modelling guidance. In *ABZ 2010.* Springer-Verlag, 2010.

[81] D. Jackson. *Software Abstraction: Logic, Language, and Analysis.* MIT Press, Cambridge, 2006.

[82] M. A. Jackson. *System Development.* Prentice Hall, Englewood Cliffs, 1983.

[83] C. Jones, P. Hearn, and J. Woodcock. Verified software: A grand challenge. *Software Technologies, IEEE Computer Society*, pages 93–95, April 2006.

[84] C. B. Jones. *Sytematic Software Development Using VDM.* Prentice Hall International, second edition, 1990. Available from http://www.freetechbooks.com/about244.html.

[85] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. In *Verified Software: Theories, Tools, Experiments.* Zurich, Switzerland, 2005. Available from http://vstte.ethz.ch/papers.html.

[86] E. Jou and J.H. Jeppesen III. Flash memory wear leveling system providing immediate direct access to microprocessor, October 1996. US partent 5,568,423, Filed April 14, 1995; Issued October 22,1996; Assigned to Unisys.

[87] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In Börger et al. [19], pages 294–308.

[88] E. Kang and D. Jackson. Designing and analyzing a flash file system with Alloy. *Int J Software Informatics*, 3(2-3):129148, 2009.

[89] H. Krumm. Temporal logic. Technical report, Department of Computer Science, University of Dortmund, 2000.

[90] A. Krupp, W. Mueller, and I. Oliver. Formal refinement and model checking of an echo cancellation unit. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 30102, Washington, DC, USA, 2004. IEEE Computer Society.

[91] L. Lamport. The temporal logic of actions. *ACM Toplas 16*, 3:872–923, 1994.

[92] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[93] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[94] X. Liu, H. Yang, and H. Zedan. Formal methods for the re-engineering of computing systems: A comparison. Technical report, Software Technology Research Laboratory, De Montfort University, England, 1997.

[95] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83:97–130, 1991.

[96] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[97] C. Métayer, J.-R. Abrial, and L. Voisin. Rodin deliverable 3.2. Event-B language. Technical report, University of Newcastle upon Tyne, UK, 2005. Available from http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

[98] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[100] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[101] C. Morgan and B. Sufrin. Specification of the UNIX filing system. *IEEE Trans. Software Eng.*, 10(2):128–142, 1984.

[102] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[103] M. Ouimet. Formal software verification: Model checking and theorem proving. Technical Report, Embedded Systems Laboratory, Massachusetts Institute of Technology, March 2007.

[104] S. Owre, N. Shankar, and J. Rushby ans D. Stringer-Calvert. PVS version 2.4, system guide, prover guide, PVS language reference, 2001. Available from http://pvs.csl.sri.com.

[105] D. L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19:859–862, 1993.

[106] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[107] A. Requet. BART: A tool for automatic refinement. In Börger et al. [19], page 345.

[108] A. Rezazadeh and M. J. Butler. Some guidelines for formal development of web-based applications in B-method. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 472–492. Springer, 2005.

[109] M. Saaltink. The Z/EVES system. In *ZUM97: Z Formal Specification Notation*, pages 72–85. Springer-Verlag, 1997.

[110] S. Schneider. *The B-method an Introduction*. Palgrave, 2001.

[111] R. Silva and M. Butler. Supporting reuse mechanisms for developments in Event-B: Composition. Technical report, ECS, University of Southampton, 2009.

[112] R. Silva, C. Pascal, T.'S. Hoang, and M. Butler. Decomposition tool for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference, Canada*, 2010.

[113] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006.

[114] B. Tatibouet, A. Requet, J.-C. Voisinet, and A. Hammad. Java card code generation from B specifications. In *Formal Methods and Software Engineering*, pages 306–318, Berlin, Germany, 2003. Springer.

[115] P. Taverne and C. (Kees)Pronk. RAFFS: Model checking a robust abstract flash file store. In *ICFEM '09: Proceedings of the 11th International Conference on Formal Engineering Methods*, pages 226–245, Berlin, Heidelberg, 2009. Springer-Verlag.

[116] Y. Wang, J. Pang, M. Zha, Z. Yang, and G. Zheng. A formal software development approach using refinement calculus. *J. Comput. Sci. Technol.*, 16(3):251–262, 2001.

[117] J. Woodcock. ABZ call for papers on the POSIX pilot project in the grand challenge, 2007. Available from http://www.cs.york.ac.uk/circus/mc/abz.

[118] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice–Hall, 1996.

[119] S. Wright. Automatic generation of C from Event-B. In *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009.