Miguel Alexandre Lopes Oliveira da Costa Ferreira **Verifying Intel Flash File System Core**

UMinho | 2008

Universidade do Minho
Escola de Engenharia

Miguel Alexandre Lopes Oliveira da Costa Ferreira

Verifying Intel Flash File System Core

Dezembro de 2008

Universidade do Minho
Escola de Engenharia

Miguel Alexandre Lopes Oliveira da Costa Ferreira

Verifying Intel Flash File System Core

Tese de Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Oliveira

Dezembro de 2008

Anexo 2

DECLARAÇÃO

Nome

____

Endereço electrónico: ____________________________ Telefone: _____________ / _____________

Número do Bilhete de Identidade: ____________________

Título dissertação □/tese □

____

____

____

Orientador(es):

____

____ Ano de conclusão: __________

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

____

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, ___/___/______

Assinatura: ____

Verifying Intel® Flash File System

Abstract

The present thesis pretends to be a contribution to the Grand Challenge in verified software international initiative, specifically in the Verifiable File System case study. The main discipline behind this project is Formal Methods for software development and in particular formal specifications, in addition to the verification of such specifications. The author proposes different approaches to verify a VDM++ model, using Alloy Analyzer for model checking and HOL4 for theorem proving, as light and heavy weight approaches, respectively. The different methods require different languages and tools, and the feasibility of the integration of these methods is the key research goal of this work.

The base for the file system specifications is the Intel® Flash File System Core Reference Guide, that defines the architecture and API of a file system that is POSIX aware and specialized for flash memory devices. Requirements for the file system are captured in a VDM++ specification, together with two basic operations to create and remove files. Test cases are developed at this initial stage to validate the requirements. The proof obligations from the VDM++ specification are generated and translated to Alloy, to be model checked. The highest level of confidence is achieved by discharging the proof obligations through a mathematical proof. For that purpose the Overture Automatic Proof System translates VDM++ models, and the respective proof obligations, to HOL models that can be automatically verified by the Automatic Proof System using the HOL4 theorem prover.

The outcome of this project is the design of a VDM++ verification tool chain, that integrates language native tools, with tools for other languages, in order to automate extended static checking. Full automation has not been achieved, although some steps towards it have been taken.

Verifying Intel® Flash File System

Resumo

A presente tese pretende contribuir para uma iniciativa internacional denominada Grandes Desafios em Computação, com o objectivo de consolidar a área de verificação de software, especificamente no caso de estudo Sistema de Ficheiros Fiável. O trabalho aqui apresentado tem como base teórica Métodos Formais para o desenvolvimento de software, em particular especificação e verificação formal de correcção. O autor propõem diferentes abordagens à verificação de especificações VDM++, usando o Alloy Analyzer para verificação usando técnicas automáticas de exploração de estados e, o HOL4 para descarregar provas de correcção. Pretendendo com isto promover diferentes níveis de confiança, associados a diferentes níveis de complexidade do método. A utilização dos diferentes métodos sobre a mesma especificação, requerem uma adequada integração de linguagens e ferramentas, sendo que esse é também um dos propósitos deste projecto.

A especificação formal do sistema é construída com base no documento "Intel® Flash File System Core Reference Guide", onde se define a arquitectura e API de um sistema de ficheiros que respeita a norma POSIX, além de ser orientado para utilização em dispositivos de memória *flash*. Os requisitos do sistema de ficheiros são capturados em uma especificação VDM++, incluindo duas operações para criar e remover ficheiros. A especificação é inicialmente validada através de casos de teste também especificados em VDM++. Com o intuito de aumentar a confiança na especificação, obrigações de prova são geradas e traduzidas para Alloy por forma a serem automaticamente verificadas dentro de um espaço de estados finito. Um último nível de verificação, de onde se pretende obter o nível mais elevado de confiança, é a prova matemática de correcção. Este nível pode ser atingido com a utilização de uma ferramenta construída no âmbito do projecto Overture denominada Sistema Automático de Prova, capaz de traduzir especificações VDM++, e as respectivas obrigações de prova, para a linguagem do provador de teoremas HOL4. Com recurso ao HOL4 as obrigações de prova da especificação VDM++ podem ser descarregadas, sem que seja necessário conduzir a construção das provas.

Como resultado deste projecto o autor apresenta um processo de verificação de especificações VDM++ que integra diferentes ferramentas, tanto nativas como externas, com o intuito de automatizar o processo de verificação estática estendida. A automatização completa de processos não foi atingida, contudo regista-se já algum progresso nesse sentido.

Acknowledgments

To José Nuno Oliveira the most sincere acknowledgment for both the inspiration and opportunity to pursue an education in Formal Methods for Computer Science, under his close supervision. He provided the opportunities, and financial support that enabled me to participate in international conferences and benefit from the contact with other computer science researchers. With out his valuable guidance and motivation this project would not have been possible.

Many thanks Samuel Silva, colleague and co-participant in the Verifiable File System project, for his contribution in the initial file system specifications, the translation of VDM++ specifications to Alloy, and help re-constructing specifications that were lost along the way.

Many thanks to Peter Gorm Larsen for showing so much interest in the Verifiable File System project, and providing the opportunity to present it at the Fourth VDM-Overture Workshop held in Finland, during the FM'2008 international symposium. Also for his contribution with reviews of this thesis, and valuable advice.

Many thanks to Sander Vermolen for his extensive support on the Overture Automated Proof System, and HOL4 theorem prover. He showed great interest in the problems we faced in the Verified File System project proofs, and contributed with valuable insight to mechanically discharging them.

Many thanks to both Rajeev Joshi and the SRI International for their interest in the Verifiable File System project, and financial support that enabled me to present the project at the VS-Experiments Workshop held in Canada, during the VSTTE'2008 conference.

Finally, many thanks to the CSK Group for providing access to the full fledged VDMTools, and to Intel Corporation for the permission to reproduce parts of the Intel® Flash File System Core Reference Guide.

Contents

1 Introduction **1**
1.1 Formal Methods 4
1.2 Grand Challenge 6
1.2.1 Verifiable File System 7
1.3 Software Verification 8
1.3.1 Testing 10
1.3.2 Model Checking 11
1.3.3 Mathematical Proof 12
1.4 Objectives 12
1.5 Document Structure 13

2 Tool Background **15**
2.1 VDM 15
2.1.1 VDM++ Language 15
2.1.2 VDMTools 16
2.1.3 Overture 18
2.2 Alloy 19
2.2.1 Alloy Language 19
2.2.2 Alloy Analyzer 20
2.3 HOL 21

3 Development Process **25**
3.1 Abstract Modeling 25
3.2 Model Translation 26
3.2.1 VDM++ to Alloy Translation: Hand Guide 26
3.2.2 VDM++ to HOL Translation: Preparation 31
3.3 Verification 34
3.3.1 Testing 35
3.3.2 Model Checking 36
3.3.3 Proof of Correction 36
3.3.4 Tool Chain 36
3.4 Summary 38

4 Intel® Flash File System Core 39

4.1 FS_DeleteFileDir . . . 40
4.1.1 Requirements Analysis . . . 40
4.1.2 VDM++ Model . . . 42
4.1.3 Unit Testing the VDM++ model . . . 48
4.1.4 Alloy Model . . . 50
4.1.5 Model Checking the Operation with the Alloy Analyzer . . . 55
4.1.6 Model Checking VDM Proof Obligations with the Alloy Analyser . . . 57
4.1.7 VDM++ Adapted for the VdmHolTranslator Tool . . . 62
4.1.8 Correcting the Translated HOL4 Model . . . 64
4.1.9 Discharging VDM Proof Obligations with HOL4 . . . 66
4.2 FS_OpenFileDir . . . 67
4.2.1 Requirements Analysis . . . 67
4.2.2 VDM++ Model . . . 70
4.2.3 Unit Testing the VDM++ Model . . . 75
4.2.4 Alloy Model . . . 80
4.2.5 Model checking the operation with the Alloy Analyzer . . . 87
4.2.6 Model Checking VDM Proof Obligations with the Alloy Analyser . . . 90
4.2.7 VDM++ Adapted for the VdmHolTranslator Tool . . . 101
4.2.8 Correcting the Translated HOL4 Model . . . 103
4.2.9 Discharging VDM Proof Obligations with HOL4 . . . 104
4.3 Summary . . . 105

5 Related Work and Conclusions 107

5.1 Related Work . . . 107
5.1.1 Verified File System . . . 107
5.1.2 Using Alloy as a Complement for Other Methods . . . 108
5.2 Conclusions . . . 108
5.2.1 Contributions . . . 108
5.2.2 Difficulties . . . 109
5.2.3 Future Work . . . 111

A Libraries 121

A.1 Alloy Relational Calculus Library . . . 121

B Models 125

B.1 VDM++ Model . . . 125
B.1.1 Flash File System Core . . . 125
B.2 Unit tests . . . 133
B.2.1 FS_DeleteFileDir . . . 133
B.2.2 FS_OpenFileDir . . . 137
B.3 Alloy Model . . . 144

B.3.1 Flash File System Core 144
B.4 Model Checking 158
B.4.1 FS_DeleteFileDir 158
B.4.2 FS_OpenFileDir 160
B.4.3 Proof Obligations 166

List of Figures

1.1 Flash File System Core Components. 9
1.2 Basic Allocation Layer Components. 10

2.1 VDMTools Workbench. 16
2.2 Alloy Analyzer Workbench. 21
2.3 Alloy Model Instance. 22
2.4 HOL4 Interpreter. 24

3.1 Preparation of VDM++ Models for the VdmHolTranslator 33
3.2 VDM — Alloy — HOL Tool Chain 37

4.1 `FS_DeleteFileDir` Operation. 40
4.2 `FS_OpenFileDir` Operation 68

5.1 Alloy — VDM — HOL Tool Chain 110

List of Tables

4.1 Abstracted fields from structure FS_FileDirInfo 44
4.2 Abstracted fields from structure FS_OpenFileDir 45
4.4 Test Coverage for Functional FS_DeleteFileDir 49
4.6 Test Coverage for Objectified FS_DeleteFileDir 49
4.8 Test Coverage for Functional FS_OpenFileDir 77
4.10 Test Coverage for Objectified FS_OpenFileDir 77

Acronyms

API Application Programming Interface . . . 7

APS Automatic Proof System . . . 18

AST Abstract Syntax Tree . . . 18

BAL Basic Allocation Layer . . . 8

BDD Binary Decision Diagram . . . 11

CICS Customer Information Control System . . . 4

CORBA Common Object Request Broker Architecture . . . 16

CPO Complete Partial Order . . . 23

CPU Central Processing Unit . . . 110

CSP Communicating Sequential Processes . . . 25

DOL Data Object Layer . . . 8

FIFO First In First Out . . . 40

FIL Flash Interface Layer . . . 8

FSL File System Layer . . . 8

FTL Flash Translation Layer . . . 7

GC Grand Challenge . . . 6

HOL High Order Logic . . . 12

IFFSCRG Intel® Flash File System Core Reference Guide . . . 8

ISO International Organization for Standardization . . . 15

JFFS Journalling Flash File System . . . 7

LCF Logic for Computable Functions . . . 21

LL Low-Level . . . 8

LSM Logic of Sequential Machines . . . 23

MIT Massachusetts Institute of Technology . . . 107

ML Meta Language . . . 21

NASA National Aeronautics and Space Administration . . . 5

ONFI Open NAND Flash Interface 8

PO Proof Obligation 17

POSIX Portable Operating System Interface 6

PSBC Pseudo Single Bit per Cell 42

PVS Specification and Verification System 12

RAM Random-Access Memory 8

RM Reclaim Module 8

SAT propositional satisfiability 11

UML Unified Modeling Language 4

VDM Vienna Development Method 4

VDM-SL Vienna Development Method — Specification Language 4

VDM++ Vienna Development Method — Object Oriented Language 4

VFS Verifiable File System 7

VICE VDM++ In Constrained Environments 4

VSR Verified Software Repository 6

VSTTE Verified Software: Theories, Tools, Experiments 6

YAFFS Yet Another Flash File System 7

Chapter 1

Introduction

Computing machines have taken control of most aspects of current life. In the beginning they were mechanical and created to replace people who performed calculations by hand. The mechanical components of the machines evolved to electric components which became smaller, performed better and consumed less power. The evolution continued with the digital era, were electronic components brought the same kind of benefits, although in a much greater scale.

With the goal of automating calculation, the first computers were single purpose, or single program, machines completely built in hardware. Charles Babbage, an English mathematician, philosopher and mechanical engineer, started to work on a mechanical calculating machine to compute polynomials in 1822 [94, 11], that he called Difference Engine. Although his first attempt to build such machine lasted more than 10 years, he failed to completely deliver it. Before his death Babbage would develop two other models for computing machines, one called Differences Engine No. 2 (successor to his first machine), and another called Analytical Engine capable of calculating any function. The later was a fully programable model for a machine powered by a steam engine. Babbage had the idea of allowing user programs to run in the Analytical Engine through a mechanism to access strings of paper cards that would contain the operations or the variables necessary for the execution. At this time he had already the idea of separating instructions from data, based on his background on algebraic notation and in his conviction that there should be a clear separation between operation and quantity symbols. The instruction set Babbage devised included operations to move card strings back and fourth, allowing for loops in the programs.

In 1936 Alan Turing presented an abstract computation model [87], later called Turing machine, capable of describing the logic behind any computational algorithm. Turing proposed in this model that the machine could execute any arbitrary sequence of well formed instructions. The Turing machine was never built, although the study of its abstract properties added to the knowledge on computing science. Scientists have always relied on abstraction to clear irrelevant details, hence decreasing complexity of the observed phenomena. Later on (1945), another mathematician and computer scientist, John von Neumann, wrote an incomplete document [93] (published in 1993) describing a computing machine architecture, that complied with the ideas and principles of the Turing machine. The von Neumann architecture is the ancestor of the computing machines we have today, it was the first specification of a concrete machine that would fit the current notion of

computer. Since then, hardware has evolved in a sound and sustained manner, where by increased complexity has brought huge performance benefits, instead of dramatically increasing design flaws. The fact that the machines we call computers have evolved from research made by mathematicians, using tools such as abstract models based on mathematical logic, in the author's perspective, is no coincidence. An incident such as the famous FDIV bug in Intel® Pentium® [64], with a quantified cost of over $400 million [29], was enough for hardware industry to look for better validation strategies, especially those able to find design flaws. Increased success of formal verification of hardware components has convinced engineers that using rigorous mathematical based approaches can in fact be cost and time effective. In 1992 Edmund Clarke et al published a paper [19] reporting on the verification of the Futurebus+ cache coherence protocol, where they discuss the results of formalizing the protocol, and point out some errors found in some possible configurations. Errors were also found in the Scalable Coherence Interface protocol [85].

The advent of calculation automation and general purpose programable machines, opened up new domains where computers could be used. Programmers could now use the same machine to execute different code, thus increasing the overall utility of computers. However, ten years ago, it would be difficult to imagine that computing machines would be so widely spread, hidden from your eyes, controlling almost everything, from the most vital sectors of society, to the tiniest device we use to hear music. Software related technologies have expanded very fast, although their influence and importance has expanded even faster. As hardware became more stable and mature, software became unreliable and messy. The common practice among programmers is to write code to do some task, and then write some test code in order to see if it produces the desired output by means of simulation. When it does, then it's considered to be working! This testing approach has two major fallbacks, (1) as complexity increases the domain space of inputs to test increases dramatically, (2) sometimes the domain space of inputs tends to infinite, it often stops only at the physical limitations of the machine. The point is that, although systematic testing is very important, it simply is not enough.

The need for a different approach to software development and verification, producing better results than traditional validation is consensual among software engineers in areas where systems are considered to be life-critical or mission-critical. In these areas, such as avionics, nuclear reactor controllers, human medical care, and transportation management systems, unpredicted software failures can be disastrous, sometimes at the cost of human lives. There are some good examples illustrating how software failure leads to disastrous results [13]:

ARIANE 5 rocket launcher exploded on June 4 1996, with a total loss over $850 million.

Therac-25 a computer-controlled radiation therapy machine, between June 1985 and January 1987, massively overdosed six people, killing two.

Denver Airport's computerized baggage handling system delayed opening of the airport by 16 months, and the overall cost was $3200 million over budget.

As Ricky W. Butler et al so well put it in article [14]:

> *Unlike physical systems that are subject to physical failure, in software, there's nothing to go wrong but the design.*

Why is it so difficult do develop correct software? Why is software so different from other systems that have been built in the past with great success? In comparison with the hardware industry, David Dill et al [29] suggest that the application of formal technics by this industry have been cost-effective, thus the acceptance it has gained among hardware engineers. Furthermore, formal hardware verification has become attractive because of methods and tools aimed at finding bugs, instead of trying to assure perfection. In the article the authors make the case that, although hardware is quite sophisticated and complex, it does not deal with pointers, possibly infinite loops and recursion, or even dynamically created processes. So, in their perspective, software is a bit more laborious to formalize and verify although the same principles could apply, as in many other engineering disciplines.

The difficulty in applying traditional engineering validation technics, such as predicting behavior and values through calculation, lies in the characteristics of discrete systems. Traditional engineering deals with continuos systems, where a small change in the input value usually produces a small change in the output value. Software engineering has to deal with discrete systems, where a small change in the input can yield great changes on the output. Furthermore traditional engineering is conservative and rigorous in the application of science, in opposition to software engineering which is opportunistically ruled by ever changing tendencies. Because a software program can easily present many millions of discrete state transitions, about which designers must reason in a way that allows the prediction of the behavior without having to test every single transition, rigorous development base theories, methods and tool must be mature enough to elegantly handle such complexity. Testing is a common practice in software development, although as complexity increases the amount of time necessary to conveniently test software turns it an unfeasible approach. A good example is found in the avionics area: *"to measure a 10^{-9} probability of failure for a 1-hour mission one must test for more than 10^{9} hours (that is to say, 114,000 years)"* [14]. Another way that industry uses to deal with design flaws is design-diversity, where different implementations from the same specification are used in parallel, and hopefully erroneous outputs will be discarded through voting processes. This practice is based on the assumption of independent faults, which is rejected for low reliability systems, and difficult to validate for high reliability systems. Because testing is unfeasible, and design-diversity cannot provide the necessary reliability the alternative is to avoid design flaws in the first place. The differences between classic continuous systems and discrete software systems, and the different ways to reduce software faults are discussed in detail in [15].

1.1 Formal Methods

> *"Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (...) The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs.* [13]

Formal Methods in computer science are the applied mathematics of software, based on the same mathematical logic as other more mature engineering disciplines. However, because other areas of science already have a greater body of knowledge, scientist and engineers do not need to express their problems and solutions directly on simple logic notation. Even though software engineers can resort to reusable libraries, generic operating systems routines, and software modules, building an application is always building something new and in many cases with unique and implicit properties, that need careful and rigorous analysis every time.

It is a fact that the vast majority of software engineers does not use mathematical logic based reasoning to design their programs, even though mathematics is what sustains all true engineering practices. Why so much resistance to engage in rigorous mathematical processes? Is rigorous a bad thing when applying engineer concepts? Formal Methods critics usually point the complexity and difficulty of the methods and techniques as the major drawback in its application. On other hand, the success achieved by the use of Z [81] modeling language (ISO/IEC 13568:2002) in the upgrade of the IBM Customer Information Control System (CICS) software to its version 3.1, is an example of how much one can save in development, deployment, maintenance and upgrade of software. In the case of CICS the authors claim that the user reported errors dropped down to half the normal amount, and a 9% savings in the total development cost [31] ($13 million [15]). Another negative claim about Formal Methods is that the formal development and verification processes consume much more time than traditional methods. This claim has been argued against by people that have successfully applied some formal methods together with other widely used informal methods. [89] presents such a success case, where Vienna Development Method — Object Oriented Language (VDM++) and Unified Modeling Language (UML) were combined to specify and guide the implementation of a distributed real-time auctioning system, while [88] provides an example of a industrial fixed price, fixed date project where Vienna Development Method — Specification Language (VDM-SL) was used, with success, in the development of a data centric application.

The state of the art of Formal Methods tools for software development is sometimes insufficient to accurately describe some complex systems' behavior. There are advanced efforts in introducing concurrency, synchronization and hardware capabilities and configurations in the Vienna Development Method (VDM) community, with the VDM++ In Constrained Environments (VICE) ex-

tension [90, 61]. Extending modeling languages and tool sets is a valid option to cover more ground on formal specifications. In this MSc project a different option was taken, consisting on using different modeling languages to capture different aspects of the requirements, with different abstraction levels and different expression power (as proposed in [21]). The use of different models of the same system is widely used across multiple fields of engineering, such as building skyscrapers, bridges, plaines, cars, computers and so on. It is not hard to understand the reason behind the use of different models in the development of a car. For example, to test aerodynamics in a wind tunnel engineers can abstract many if not all mechanical details, that would not be abstracted if testing the car's behavior on aquaplaning situations. With different kinds of models and abstractions come different set of formal tools that can be used to reason about, verify and better develop software. Integrating heterogeneous tools to achieve greater insight on the requirements, implementation, and verification is one of the main lines of research of this project.

Holloway, from National Aeronautics and Space Administration (NASA) Langley Research Center, presented a paper [17] more than ten years ago, where he analyses why software engineers do not do follow the same mathematical approach as do engineers from other fields of science. In the paper, Holloway questions if the problem in the poor acceptance of Formal Methods is related with how its benefits are communicated to software engineers, and suggests an alternative argument, that he shows to be simpler and stronger. His argument is based on very simple inference rules:

> *Software engineers strive to be true engineers (Q1); true engineers use appropriate mathematics (Q2); therefore, software engineers should use appropriate mathematics (Q3). Thus, given that formal methods is the mathematics of software (Q4), software engineers should use appropriate formal methods (Q5).*

The term "appropriate formal methods" takes us back to the tool integration topic, because not every formal method is appropriate for every situation, and sometimes only through the combination of different methods the desired insight, confidence, and correction can be achieved. It also means that formal specifications are useful whenever the abstraction level is just enough so that irrelevant implementation details are overlooked, and all key aspects are taken in account. Choosing the appropriate abstraction for each specification is not trivial, and has a great impact on the quality and usefulness of a formal model.

Formal Methods have been associated with academic research for a long time. In academia theories grow sounder, process and techniques have been developed and applied to case studies as proofs of concept, and many tools have emerged. The current maturity of Formal Methods makes it possible to aspire for knowledge transfer to industry applications. As long as scientific products remain in research they will always be in development and continuous research cycles, although whenever they slip to industry, tools tend to become stabler, more mature and specialized. It is necessary to access if the available Formal Methods scale up to real size systems, integrate with other methods used in software engineering, can be used in a cost effective way, and provide quality tools. This leap to general industry applications is of great importance to the effectiveness of Formal Methods. The NASA Langley's Research and Technology-Transfer Program in Formal Methods [13]

aims for such a transfer of knowledge to industry. The Grand Challenge (GC) is another example of effort to bring Formal Methods closer to application in real software, as explained bellow.

1.2 Grand Challenge

At the Verified Software: Theories, Tools, Experiments (VSTTE) conference in 2005, Hoare and Misra [41] argued that it was time to embark on a international GC project to construct a program verification tool that would increase the capabilities of today's static checkers, based on mathematical logic, that would be able to automatically check the correctness of a program. The tool set would have to be based on a sound and complete programming theory. The proposal is quite ambitious, being that the GC project would span over 15 to 20 years of scientific research, it should be a major international research effort with specific measurable goals: one million lines of verified code, together with specifications, design, assertions, etc. The proponents expected that the initiative would consume over one thousand person-years of skilled scientific effort, from all over the international scientific community, in the sense that each researcher would give a contribution that would be shared with all the community. The project would yield its results in three categories: theory, tools and experiments. It expects to achieve a unified theory of programming, that would cover the majority of paradigms and design patterns in real programs; an integrated tool set for design, development, analysis, verification, testing, upgrading and automated generation of programs; a repository of verified software to hold realistic case studies and tools. The GC experiments would start with smaller pilot projects (1-5 years) that would gather evidence of viability of the long-term goals, and would be a way to attract scientists to the project.

This visionary view of challenging the computer science community, in order to stimulate scientific development, is proving to be a success as the number of researchers, and research groups, that claim to be motivated by the GC initiative increases every year.

One example of organizational effort in increasing computer science research, in a GC alike fashion, is the Grand Challenge in Computer Science [22] project, an enterprise of the United Kingdom Computer Research Committee. Of particular interest to this thesis is the "GC6: Dependable Systems Evolution", in which the Verified Software Repository (VSR) [6] was created, a repository for specifications and programs, to be used for development and testing of tools, that themselves are being collected in the repository. Bicarregui et al [7] believe the VSR will, among other things, contribute to the inter-working, and eventual integration, of formal tools.

Since the "idea" of GCs as a stimuli for research on Formal Methods was proposed, many challenges, typically 1 to 5 years, have been produced by the scientific community. For example, the Mondex case study [96], the Pacemaker challenge [51, 54], and the Portable Operating System Interface (POSIX) File Store [33]. Accompanying this trend for challenges, Joshi and Holzmann [47] proposed a mini-challenge, 2 to 3 years of work, for building a verifiable file system. This proposal was motivated by a major fault in the Flash memory subsystem of the Mars Exploration Rovers [73]. NASA engineers successfully overcame this fault, although the affected robot lost a large portion of its data storage capabilities, reducing its autonomy to perform science.

1.2.1 Verifiable File System

The Verifiable File System (VFS) "mini-challenge" [47] is a short-term version of a GC project. It is an introductory step for a repository of formal models, tools, and software, as proposed in [41]. The proponents consider that reducing the challenge complexity, hence the "mini-challenge", would be a benefit to the agreement on common formats and forging the necessary collaborations to build such a repository. Furthermore, they believe that a file system is a good candidate because there is a clean and well-defined interface, known as POSIX [82, 83, 84], the data structures and algorithms for a file system are well understood, in addition to the fact that a file system is complex enough so that its formal modeling and verification are not trivial. The target hardware for the file system are Flash memory devices, which have many design and behavior differences from spinning disc block devices. The main goal is to build a formal specification of a POSIX compliant file system that would enable the use of automatic verification techniques, in order to assure its correction. The formal specification could also be used to verify existing and future implementations. It should include:

- *a formal behavioral specification of the functionality provided by the file system*;
- *a formal elaboration of the assumptions made of the underlying hardware; and*
- *a set of invariants, assertions, and properties concerning key data structures and algorithms*.

The specification should provide ways to ensure reliability regarding concurrent access, unexpected power loss, and hardware failure.

Before the POSIX standard was even published, Morgan and Sufrin [60] wrote a Z specification of the UNIX filing system, capturing the behavior at system call level, and abstracting from data representation details. It was later used by Patrick Place in the development of his Z specification of the POSIX Application Programming Interface (API) [70, 74]. The actual informal POSIX specification provided by Open Group used Place's formal specification as a guideline. Formalizing the POSIX in to a machine tractable language is probably the most common approach to the "mini-challenge".

Flash memory production is developing fast in terms of miniaturization, power consumption, and storage capacity. It is very attractive for space exploration missions because it has no moving parts, and is easily embedded in very small and mobile devices. As capacities enlarge it is now making its path to portable and personal computers. There are two kinds of Flash memory, NOR and NAND, the second dominating the market at the moment, obviously because it is cheaper, it has greater density, and better overall performance [5, 52]. On the NAND down side it must be written one page at a time, erased one block at a time, and it does not directly support execution of code in place. Another significant characteristic of Flash memory is that its cells wear out along time, that is to say that there is a maximum of erase-write cycles they can survive. There is also the problem of power loss, or memory device removal from host bay, which should also be considered.

Journalling Flash File System (JFFS) [98] and Yet Another Flash File System (YAFFS) [67] are two popular examples of file systems designed according to Flash specificities, although in many cases, regular block device file systems are used with a Flash Translation Layer (FTL), that emulates a block device on top of the Flash device. Furthermore, advances in the devices' technology

have introduced a Block Abstracted mode [99], where all data are assessed by a logical block of granularity of a sector.

Rather than model the POSIX interface, in this project the base file system informal specification will be Intel® Flash File System Core Reference Guide (IFFSCRG) [24] document. This is an API of a flash memory file system. During the project some inconsistencies on the operations return status data type were found, contacts have been made with Intel Corporation, although with no practical result because the document was discontinued. However, copies can be obtained through the Intel Museum [23]. Regarding the necessary assumptions about the underlying flash hardware behavior a good candidate for a specification is the Open NAND Flash Interface (ONFI) [99], that is a standardization effort made by a consortium of device manufacturers. There is already an attempt to formalize this specification done at Minho [28] during an undergraduate formal methods class, where the memory organization, and device architecture was formalized in VDM++. More work on the formalization of the ONFI specification can be found at [16, 48]. Building formal models of both textual specifications would provide an adequate base for formal analysis, and proof of correction, of the file system.

Intel® Flash File System Core

The IFFSCRG document defines a layered architecture (see Figure 1.1) where each layer perform at different conceptual level, although they are related to each other as one monolithic file system. Each layer's API is defined, as well as a few examples of the work flow involved in a top level API call. The top layers, File System Layer (FSL) and Random-Access Memory (RAM) Buffer, interface with the operating systems while the lowest layer, Flash Interface, interacts with the device driver. Within the architecture, each layer should provide functionality to the next upper layer, and delegate work on the next lower layer. The Reclaim Module (RM) is the exception, interfacing bidirectionally with the Basic Allocation Layer (BAL), that stands at the same conceptual level.

The FSL exposes, to the operating system, the basic operations of the file system, so that applications can manipulate files. It depends on the Data Object Layer (DOL) to provide an abstraction of file system data as data objects. The DOL maps data objects in logical units, that are managed by the BAL. BAL is defined as a sub-architecture (see Figure 1.2) whose job is to manage logical units and map them into physical units. The RM is used by BAL to recycle dirty space produced during the system's operation. In order to perform the cleaning tasks it depends on BAL to manage logical units. The Flash Interface Layer (FIL) interfaces with the platform specific Low-Level (LL) driver, translating the system volume operations into driver calls.

1.3 Software Verification

Mathematical verification is the only feasible way to assure that any given property holds for every given input, or that some kind of undesired situation never occurs, whenever dealing with input spaces whose size tends to infinite. The main differences to the simulation or testing techniques are the ability of verifying each and every input without the need to actually compute every possibility,

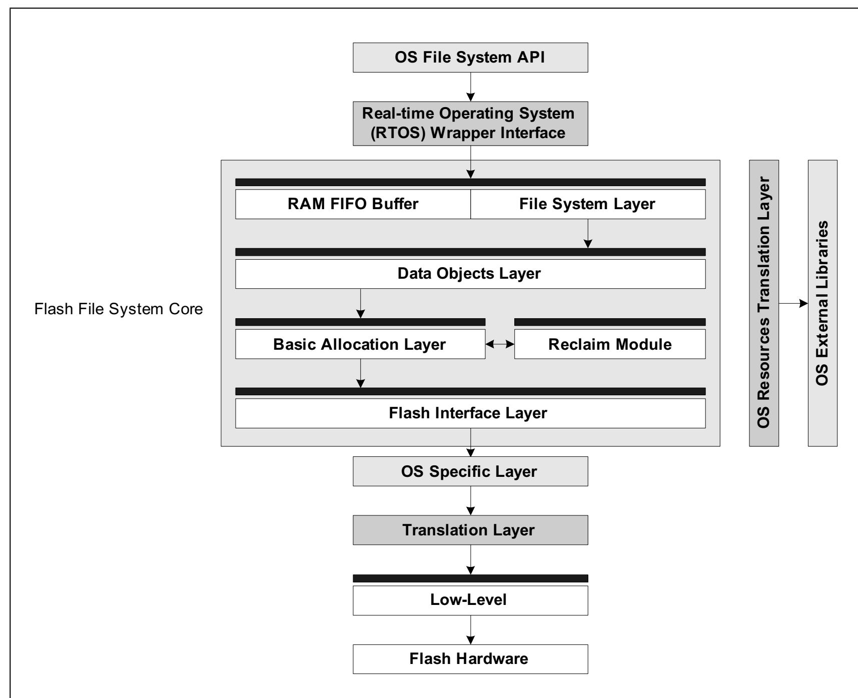

Figure 1.1: Flash File System Core components (Permission to reproduce this excerpt is kindly granted by Intel Corporation).

and the degree of confidence each provides, being that the highest level of confidence possible can only be achieved by a mathematical proof of correction. The software industry, or at least the companies that are more committed to quality, spend a great amount of money in certifying development processes. However it is not assured that correct, or reliable, software will be produced with such processes. It is reasonable to think that a better development process, should produce a better product. However, to establish the quality of a product direct analysis of the product should also be considered. So, only software verification can ensure quality. The greater lesson that the Formal Methods community can learn from the hardware industry [29], is that reaching perfection in terms of correction and reliability is not the goal for the average software development company. It is not what they strive for, because in many cases there would not be a consequence to the company if their software malfunctions. In fact it is very common, and profitable, to release defective software, and then, spend years releasing updates and patches and earning money out of maintenance contracts. Assuming that whatever Formal Methods can do for software engineers, it has to be cost-effective. Fact is that almost every piece of software has bugs, and that great amounts of money are spent in testing, updating, and correcting bugs, so it seems reasonable that if Formal Methods can find bugs, it will lead to greater acceptance by industry.

There are different levels of confidence that can be established, it can be that we assure a correct behavior for specific inputs, for a specific space / scope of inputs, or for the entire set of state input spaces.

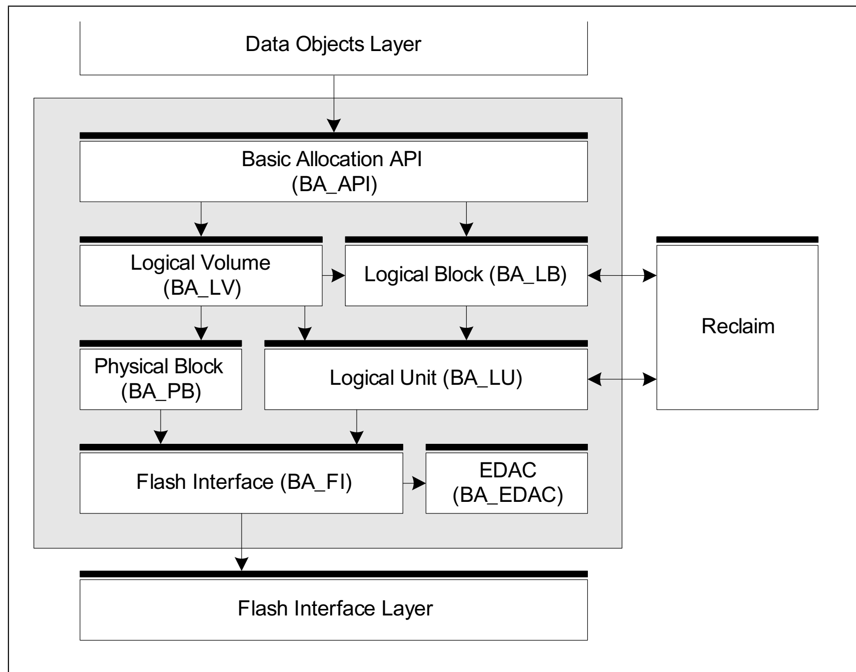

Figure 1.2: Basic Allocation Layer components (Permission to reproduce this excerpt is kindly granted by Intel Corporation).

1.3.1 Testing

Software testing is the most widespread technique to validate software, and to demonstrate to stakeholders that the results are within the expected. It is essential to software engineers, allowing them to observe the behavior of the software they produce and assert if it is performing as expected, then identifying possible malfunctions. Testing has ben targeted by numerous researchers [44, 3, 57, 35, 77] to make it more effective, reliable, and increase its scope of application. It is common practice for software development companies to hire people specialized in testing, and quite often even assemble testing teams separate form the development teams. There are even companies in which the main commercial activity is to test software developed by others. So it seems fair to say that software testing is already a discipline on its own. The field as evolved deeply, from simple *ad hoc* test code to systematic test and input generation, different criteria have been identified according to both the software under test, and the objectives of the tests. The diversity brought the possibility to conjugate different kinds of tests in different stages of the development process, starting at early stages to gain confidence in specifications and to discuss requirements with domain experts.

Tool support for testing is mature enough to be seemingly integrated in the cutting edge development tools used in industry. There are complete frameworks for large scale software testing, focusing on diverse aspects of programs: structure, data, functionality, integration, communication, scalability, robustness, and security. A current extreme programming practice Test Driven Development advocates automated test should be written before writing a specification or implementation. The analysis of test runs is another area of tool specialization where graphical displays of execution

traces plays a major role.

Within the scope of this project, testing will take place after the formal modeling of requirements, with the objective of assessing that the specification expresses what is meant. This is possible only through the use of executable formal models, and the choice here is VDM++ modelling which allows for both implicit and explicit behavior definition, being that the later can be executed in the VDMTools interpreter. The VDMTools are particularly attractive as a test bench due to their test coverage and pretty printing features, and especially to the Dynamic Type Checker that can continuously check for pre- and post-conditions, correct partial function applications, invariant preservation, valid variable bindings, satisfiability, and more. Tests will be written for specific functions as unit tests and automated through the VDMUnit [32] framework.

1.3.2 Model Checking

Model checking consists of exhaustively verifying that an asserted property holds for a defined input space. This technique can establish mathematical correction within the scope of operations, that is to say it can assure that for the checked input space the software always behaves correctly. Model checking is based on temporal logic, where as time goes by the value of a formula can change. This makes model checking very suited for checking properties over transitional systems [18, 20]. Such as many other software verification techniques, it started to be applied to hardware verification, with great success. However in the case of software even the a very simple artifact often has an infinite set of states, and to make model checking feasible it is necessary to have both optimization and abstraction techniques to reduce the input state space. Binary Decision Diagrams (BDDs) were a major breakthrough in reducing the number of states that it is necessary to check. Symbolic model checking [12, 95] appeared as another attempt to reduce the state space, by representing states symbolically instead of explicitly. Both BDDs and symbolic model checking can be used together with propositional satisfiability (SAT) [49, 56] solvers, to obtain even better results. SAT solvers are particularly interesting because they easily provide counter-examples, that can be mapped from boolean formulas to a particular state in the specification, or program.

Due to its exhaustive search nature, model checkers are very useful for detecting subtle design flaws, that are often overlooked by testing. Whenever an assertion fails, the model checker precisely identifies (sometimes graphically) the inputs that originated the failure. These inputs are often converted to test cases that can detect such situations during test time. This formal method can be very useful, allowing to uncover bugs, as well as the causes and the consequences of the bugs. It is the most wide spread formal method in the hardware industry, and responsible for discovering some of the greater bugs found so far. However model checking can only assure correction of software in very special cases, where the state space is finite and computationally tractable. In most cases, model checking can not assure the correction of a software artifact, because the large input states space would require huge amounts of computational resources or time to process a graph with all states. This is the well known *state explosion* problem.

Examples of model checkers are SPIN [42] for distributed and asynchronous software systems, CHESS [62] for the verification of multithreaded software, and Alloy [43] as a formal specification

language that can be automatically checked by the Alloy Analyzer.

1.3.3 Mathematical Proof

Because of the state space explosion problem, none of the above mentioned formal methods, testing and model checking, can verify that a software artifact is 100% correct. A way to overcome this difficulty is through a mathematical proof, that establishes beyond doubt that the desired propriety holds for any given input. This is the strongest formal technique in terms of correction, because it achieves the topmost level of confidence that there can be. It is also the most complex, and the one that requires more expertise.

Proofs can be built by hand [59, 45], using mathematical theory as a base to develop a specific theory related to the software artifact that is being targeted by the proof. The trend in software correction proof has been the semi automated proof, using interactive theorem provers, where the user chooses the theorems the machine must use, step by step. This semi automated method is also used to confirm the correctness of the hand made proofs, because it is less error prone. The ideal situation would be such that theorem provers could use all mathematical knowledge and exhaustively search proofs for properties, although this would also lead to break the time and resource constraints that software development faces these days. None the less, completely automated proofs are possible, and can be efficient. In order to completely delegate the proof to a theorem prover that can effectively discharge the proof, it is necessary to supply an adequate strategy for the theorem prover to follow. Strategy consist of restricting theories, defining domain specific theorems, and specifying the paths that the theorem prover should follow to discharge the proof, that is to say, if it should try rewriting followed by simplification and then induction steps, or if that fails it should try a different path.

Mathematical correction proof usually requires greater knowledge of the proof theories, either to manually build the proof, guide a semi automated theorem prover, or write a proof tactic for a fully automated theorem prover. This technique can also give feedback on the targeted properties by deriving the boolean value False from them, or just stopping on an intermediate goal that can give some insight on the property. Building a theory of software and automating the proofs is one of the main research areas on Formal Methods. Examples of theorem provers that are commonly used to verify software are Z/Eves [76], Specification and Verification System (PVS) [69], Coq [4], and High Order Logic (HOL) [36, 80, 65].

1.4 Objectives

Out project has focus set on integration of different formal methods for verification of software to different degrees of confidence. Based on the correction by construction principle, software implementation should be a result of previous abstract formalization followed by gradual reification. The present thesis tries to contribute to the development and automated verification of VDM++ models through the integration of different tools and languages. We aim at the integration of a test framework, a model checker, and a theorem prover, resorting to language translations between VDM++,

Alloy, and HOL to assemble a verification tool chain.

We are also interested in the VFS "mini-challenge" as a case study whereto experiment with the verification tool chain. The specification effort has focus set on the FSL, presented in the IFFSCRG document, and consists in writing a formal specification of its data structures, invariants, operations, and ensuring extended static checking of such operations. The IFFSCRG document defines power loss recovery techniques at the reclaim, logical and physical units, and file system levels. These reliability aspects are not covered in the scope of this project, although can incrementally be added.

1.5 Document Structure

The next Chapter introduces the formal tools involved in this project, and should be completely, or partially, skipped if the reader is familiar with VDM, Alloy and HOL. Chapter 3 defines the proposed development process, which is amenable to formal verification. It starts with abstract modeling, continues with the different kinds of models and abstracted details, and a structuring approach to handle complexity. Reification techniques to incrementally increase details towards implementation should also be considered, although will not be covered. It also covers verification with different degrees of confidence, obtained by the integration of multiple tools in a tool chain. The models of the Intel® Flash File System are explained in Chapter 4, and the VDM++ to Alloy translation is exemplified. Finally Chapter 5 ends the document with an analysis and comparison to related work on both the VFS "mini-challenge", the use of Alloy as a third party model checker, and the concluding topics, respectively.

Chapter 2

Tool Background

This chapter introduces the languages (VDM, Alloy and HOL) and tools adopted in this project. VDM and Alloy for formal modeling, VDM for testing and prototyping, Alloy for model checking, and HOL for mathematical proof. Some of the tools presented in the sequel are not mature enough to be considered ready for production, although already display satisfying capabilities for an exercise on tool integration.

2.1 VDM

The Vienna Development Method (VDM) is a formal method for designing software systems, developed in the 1970s at the IBM's Vienna Laboratory [8, 45, 46]. VDM relies on a formal specification language (VDM-SL [71]) that is very close to pure mathematical notation, with design by contract constructs, allowing for other techniques such as refinement or proof of properties. VDM-SL is an International Organization for Standardization (ISO) standard (ISO/IEC 13817-1:1996) that was extended to the VDM++ object-oriented specification language [32]. Furthermore, VDM++ was also extended to the VDM++ VICE which supports timings, concurrency, and customizable system architectures. Both VDM-SL and VDM++ support relational abstract modeling through pre- and post-conditions, and behavioral modeling where algorithms can be specified and executed. VDM tool support is provided by the commercial VDMTools [39], and by the Overture [68] open source project.

2.1.1 VDM++ Language

The choice of the VDM++ language over its standard counterpart is closely related with the fact that it has more tools for verification available. It supports classes, that can have instance variables, which can reference objects or data type instances. Its typing systems allows for: records, enumerations, simple mappings, sequences, sets, alphanumerical types, optional type, a special token data type, functions in the mathematical sense, and state manipulating operations. The operations can be written in an imperative fashion with sequential instructions and loops.

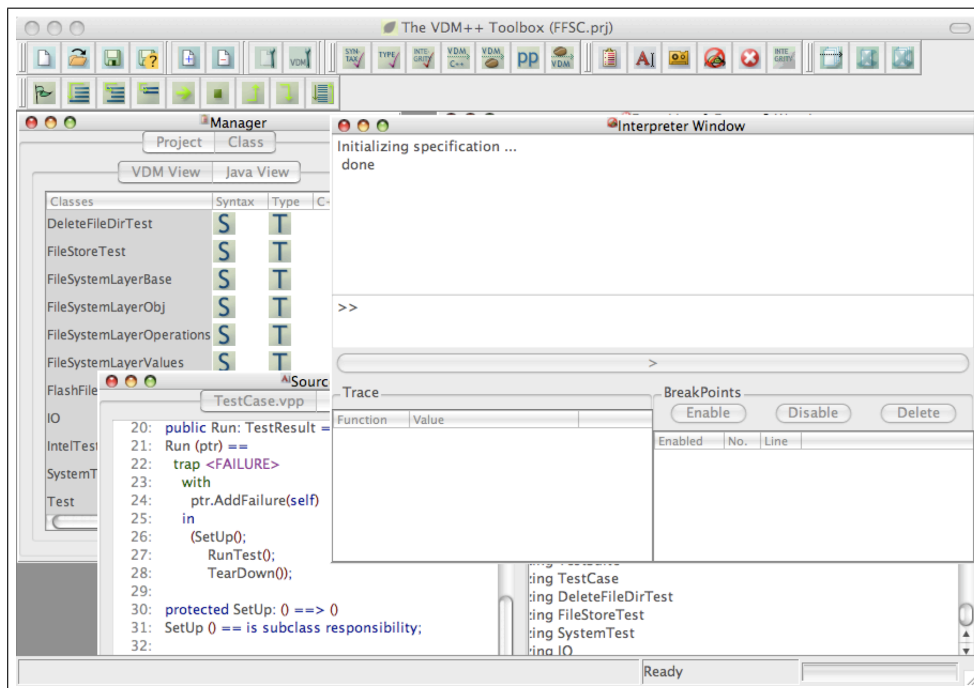

Figure 2.1: VDMTools workbench screenshot.

2.1.2 VDMTools

VDMTools (currently property of CSK Group [38]) provides both command line, graphical (see Figure 2.1 for a screenshot of the tool), and Common Object Request Broker Architecture (CORBA) interfaces. Although graphical interfaces are very powerful, through the command line, and CORBA, interfaces VDMTools expose their functionalities in a fashion that makes it easy to integrate with other tools and scripts. The main reference on formal specifications in VDM++ is book [32], where the authors extensively expose the capabilities of the language and the associated tools. It also presents a simple unit testing framework named VDMUnit, that allows for systematic testing of VDM++ specifications, much alike other unit testing frameworks. Testing specifications with the VDMTools has a great advantage when compared with other formal specification languages, and even programming languages, because together with static checkers it packs a dynamic type checker. Once activated within the VDMTools, this tool enforces invariant, pre- and post-condition checking during execution of the specification. This allows for automatic violation detection during runtime, without the need of any extra input on the specification. The tool also packs other interesting capabilities such as roundtrip code engineering, generating C++ or Java code directly from a specification, and the other way around in the case of Java. This is also very handy when targeting integration with other formal methods and tools, because it allows for VDM++ specifications to be transformed into C++ or Java for later usage within other tools. The other way around is also possible, in the case of Java, classes can be transformed in to VDM++ classes that can be used in VDM specifications.

Prototyping is another advanced feature of the VDMTools, because it provides means to execute

specifications, in addition to its CORBA interface, that allows to plug-in interfaces to the specifications. Another tool, in the package is the Integrity Checker, which generates VDM expressions, denoted Proof Obligations (POs), from a given model. The model is considered consistent if all its POs can be discharged.

POs can be of different categories:

Domain checking: POs regarding the application of functions and operators that have associated pre-conditions. These can be of types:

- Function Applications
- Mapping Application
- Sequence Application
- Non-empty Sequence
- Sequence Modification
- Map Compatibility
- Map Enumeration
- Map Composition
- Map Iteration
- Function Composition
- Function Iteration
- Non-empty Set
- Non-zeroness
- Tuple Selection

Subtype checking: POs regarding the use of subtypes, in particular types that have invariants, or are defined using the union of types. These can be of types:

- Subtype
- Invariants
- Post Condition

Satisfiability of implicit definitions: POs resulting from the use of implicit functions or operators, which must be proven total. These can be of types:

- Satisfiability
- Exhaustive Matching in Cases Expression
- State Invariants
- Exhaustive Function Patterns
- Non-emptiness of Let be such Binding

- Non-emptiness of Binding
- Unique Existence Binding
- Finiteness of Set
- Finiteness of Map

Termination: POs regarding the termination of recursive functions and loops. These can be of type Terminating While Loop.

2.1.3 Overture

Overture is an open-source community based project, where students, teachers, and researches work together at the different academic levels to develop a set of basic and advanced VDM tools. The goal is to not only provide free versions of the commercial tools for VDM, but also to develop new generation tools that take the VDM experience to the next level.

In this MSc project tools such as the Overture Parser[68] and the Automatic Proof Support [92] were extensively used (to be introduced next).

Overture Parser

As with any other language, in order to process VDM models it is necessary to have a parser that can generate an Abstract Syntax Tree (AST), here on referred as OmlAst, that can be used by other tools. The Overture Parser is implemented with the JFLEX and BYACC/J tools, allowing for integration with Java based Integrated Development Environments, such as Eclipse. The AST specification is automatically generated by another Overture tool, called ASTGEN, that is written in VDM. The actual ASTs produced by the parser, given a VDM model, can be represented as a VDM-SL or a VDM++ value, so that other VDM based tools can use it natively. The parser is the pillar that supports all other Overture tools, with it the community gained an open source AST that can be used within VDM specifications, and also in C++ and Java programs. Although the parser is a basic tool, it is of great importance and must be continuously updated to support new language constructs.

Automatic Proof Support

The Automatic Proof System (APS) is one of the next generation tools for VDM, because it offers possibilities that no other tool for VDM currently offers. It is capable of mechanically discharging VDM++ models' POs with the HOL4 [80, 65] theorem prover. To do so, it relies on the translation of the OmlAst to an HOL AST specified in VDM++. The proof system also includes some useful HOL theorems, and a set of HOL proof tactics designed to discharge VDM POs. This tool is quite recent [92] and still does not support the complete VDM++ syntax, although the supported subset of the language allows for the verification of many known VDM++ case studies, and, as it will be explained in this thesis, with a careful selection of syntax constructions it is possible to overcome many of the tool's limitations.

The one thing is that the tool is not: is automated! It involves much user input to go through the necessary steps that transform a VDM++ specification in a HOL specification, here on referred as preparation. A great deal of the user input is copy and past work, although for some steps it is necessary to select pieces and bits of OmlAst and copy them around, which can be very tricky. With some practice of using the APS it became clear that to make it worthwhile it is necessary to speed up preparation. The necessary steps of preparation will are explained in Chapter 3, together with the developed tools that further automate the process.

2.2 Alloy

Alloy is a quite recent technology, at least when compared with other modeling languages such as VDM or Z. As its authors put it [43]: *"Alloy is a lightweight modeling language for software design"*. Such as other formal modeling languages it provides a expressive mathematical notation, based on a simple logic. The Alloy Analyzer tool offers both simulation, enabling the user to request for samples of the modeled system, and checking capabilities, where it tries to find counter examples for a set of given properties. Alloy is a quite expressive language, allowing for different flavors of modeling: very abstract relational modeling (relations can even be written and composed in point free notation); simple declarative modeling, using only logic operators to express properties and behavior (much alike Z style); behavioral and temporal modeling, specifying state and transitions, and expressing algorithms through flow control with conditional statements. One of the beauties of Alloy is that all its expression power is drawn from the simplicity and elegance of the type system where every thing is a relation, and with some basic concepts such as extension, abstract data types, multiplicity and relational composition.

Alloy's model finding capabilities have already been used in a quite significant number of examples, both on its own [48, 100, 86] and as complement to other formal [9, 55] and informal methods [79].

2.2.1 Alloy Language

Alloy language is defined on very simple principles of first order logic and relational calculus. It is a general purpose language with a very simply, yet powerful, syntax. Some of Alloy's syntactic constructions and their respective semantics will be very briefly summarized below (see [43] for more details), to prepare the VDM++ to Alloy translation presented in Section 3.2.

Signatures

Types in Alloy are called signatures, although a signature can also represent a set of values. A type signature is said to be *top-level* if it does not extend any other signature. On the other hand, signatures that extend others are called sub-signatures. All the top-level signatures are mutually disjoint sets, as well as are extensions to the same signature. A signature can be declared as abstract,

which means that it will only contain the elements contained in its non abstract sub-signatures. Furthermore, signatures can have multiplicity factors associated, constraining the elements that belong to it. Signatures have fields that can be any n-ary relation, and are equivalent to having explicit facts restricting the signature.

Declarations

Fields of signatures, arguments to functions and predicates, and quantified variables are declared through the same syntax, that allows for constraining their value and type. It is possible to declare a single variable, or multiple variables at once. For the case of multiple variable declarations, Alloy allows for constraining them to be mutually disjoint or to form a partition of the relation expressed by the declaration. The elements of the relation must comply with its declared data type. The declaration of unary relations can also have multiplicity factors associated with it, and whenever these multiplicity factors are omitted, the default multiplicity is used, which is a singleton set. Sets can be declared as having an arbitrary number of elements; singleton or empty sets; or non empty sets. For relations with arity superior to one, multiplicity may be defined for each signature involved in the relation, and the same options of multiplicity apply.

Constraints

Alloy models can be further constrained using facts, predicates, functions and assertions. Facts are expressed trough formulas that must hold at all times. Predicates and functions are parametric constraints, that can have declaration of variables associated with them. Predicates have no explicit output value, and can be enforced selectively, whenever they are needed. Functions are meant for specification of computations as they can return values, although functions can not be recursive. Assertions are properties that are expected to follow from the model, and they can be checked on demand.

Commands

The Alloy Analyzer accepts commands to simulate functions, predicates or simply instantiate the model. It can also check assertions solving its constraints. To avoid the "explosion problem", in both commands there is the option to define a scope for the possible assigned values when simulating or checking. The scope defines the bounds for the sets assigned to each and every signature.

2.2.2 Alloy Analyzer

Alloy Analyzer is both a development and verification environment, as shown in Figure 2.2. With this tool it is possible to simulate and verify a specification as the same pace it is written. The tool is completely automated in terms of model finding within a specified scope, that can be both common to all signatures or individually set. Two commands are enough to verify an Alloy model, run to simulate and retrieve possible model instantiations, and check to ask for a counter-example

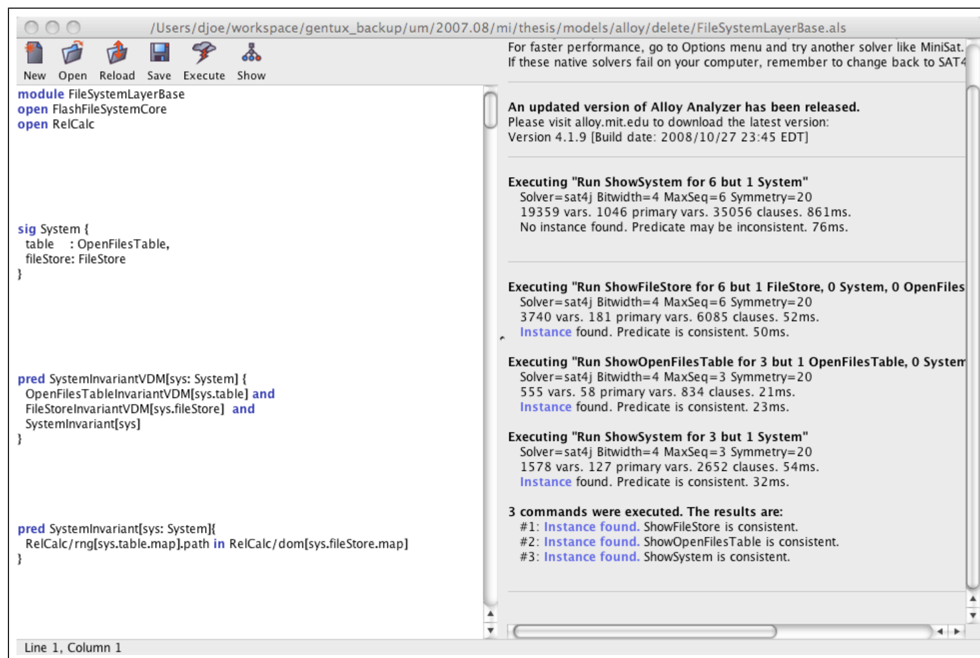

Figure 2.2: Alloy Analyzer workbench screenshot.

to a given assertion. Whenever one of the commands is issued the Analyzer converts the Alloy specification to a first order logic SAT model that is calculated by an external solver. The output of the solver is then translated back to an instance of the Alloy model that can be graphically displayed for inspection, as shown in Figure 2.3. The most significant limitation of the Alloy Analyzer is the state explosion problem that can arise as one increases the scope of objects allowed in the model instances. However it is likely that the majority if not all problems can be detected with a significantly small scope.

2.3 HOL

The HOL system is a direct descendent of the Logic for Computable Functions (LCF), invented by Dana Scott in 1969 [78] (only published in 1993), intended for reasoning about about recursive functions as defined in denotational semantics. The LCF approach lead to three versions of automated theorem provers Stanford, Edinburgh, and Cambridge LCF implementations by Robin Milner and colleagues [37]. Milner also designed the Meta Language (ML) programing language to supply a way for users of automated theorem provers to specify proof strategies, called tactics, and ways to combine the strategies, called tacticals.

The first LCF called Stanford LCF was a proof checker developed around 1972, based on Scott's logic, designed for interactively building formal proofs about computable functions, as defined in λ- $\cdot$ calculus, applied to a variety of domains. An year after Milner moved to Edinburgh University and started to work on a new version of the automated theorem prover called Edinburgh LCF. During Milner's stay at Edinburgh Mike Gordon, that would later create HOL, joined his team and got

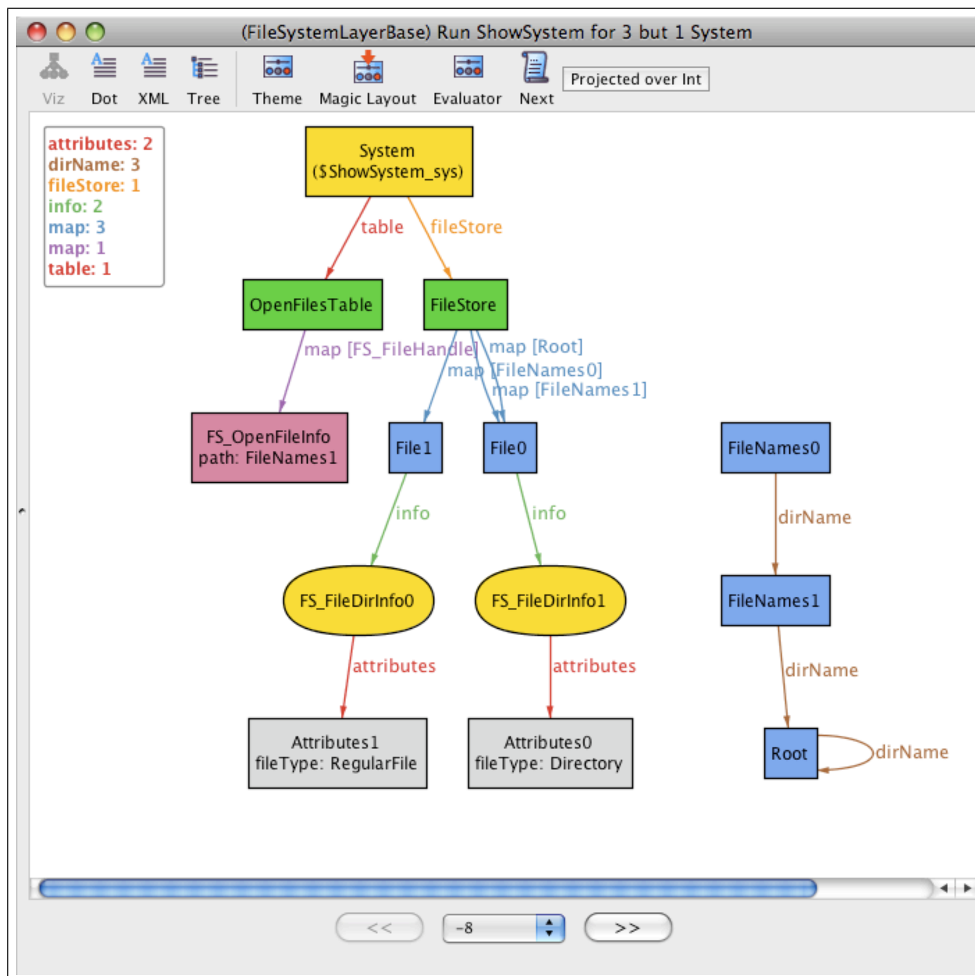

Figure 2.3: Alloy model instance screenshot.

to work in the LCF project. When Edinburgh LCF was published, in 1978, Milner had managed to limit the amount of memory needed for building a proof, and to supply a programable interface to the system through the use of the ML language. The language was made functional and supported tactics as functions that compute proofs, and tactic operators as high order functions that manipulate tactics. The soundness of this new version of LCF was established by the fact that theorems could only be added to the system by means of a formal proof. The LCF approach was followed once again in a new project, this time from a cooperation between Cambridge and Edinburgh Universities, in the 1980s, to produce the Cambridge LCF theorem prover, that was greatly extended and optimized by Gérard Huet and Larry Paulson with: improvements made to the ML language (including a compiler); inclusion of a simplifier as a derived rule; a new way to deal with indexing sets of equations used in rewriting; and the inclusion of all standard constructs of predicate calculus.

While Paulson was pushing the development of Cambridge LCF further, Mike Gordon was invested in hardware verification. He developed a notation called Logic of Sequential Machines (LSM) to express machine behavior, inspired in the **css!** (**css!**)'s Expansion Theorem. Gordon used the Cambridge LCF to implement the LSM logic in what he called the LCF_LSM, that would later result in the first HOL system. Gordon Being interested in hardware verification established that mathematical induction would be enough for the majority of the cases, removing the need for fixed-point (Scott) induction. This simplification lead to the definition of types as sets in HOL, instead of Complete Partial Orders (CPOs) as in LCF, although it still maintains much of Milner's original code. HOL is based on a well founded core logic, based on Church's simple type theory, in which terms are nothing more than variables, constants, function applications and $\lambda\text{-}\cdot$ abstractions. This way the reasoning mechanisms only need to deal with four types of terms, and therefore become simpler to specify and implement. All other complex syntactic constructions supported by HOL are defined on top of $\lambda\text{-}\cdot$ calculus. HOL supports both forward proof and goal directed proof styles, including several libraries offering different theories and proof procedures.

The HOL theorem prover has evolved deeply from its first release HOL88 from Cambridge University, through HOL90 from Cambridge University and Bell Labs, HOL98 from Cambridge, Glasgow and Utah Universities; until now with the development of HOL4 as a open source project, partially supported by the PROSPER project [27], using the Moscow ML [75] as the interface to the core logic. HOL has been used for the verification of very simple processors, simple micro processors, networking hardware, sophisticated hardware systems, and currently software. It is mainly interfaced thorough the command line as shown in Figure 2.4.

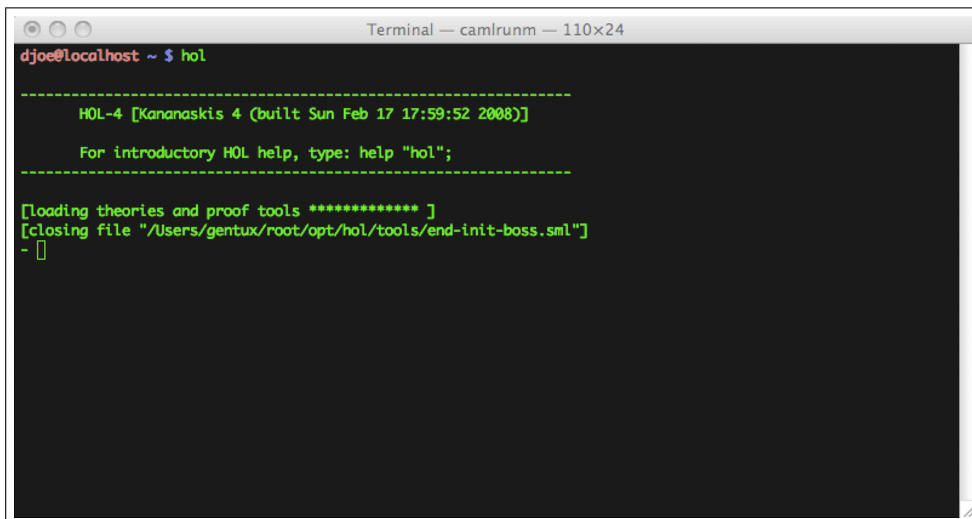

Figure 2.4: HOL4 command line interface screenshot.

Chapter 3

Development Process

To design a system it is advised to start reasoning about key principles, so that non trivial or hidden assumptions about properties and constraints of those systems can be uncovered. As the designed system's complexity increases it is necessary to deal with it in a elegantly structured fashion, so that flaws can easily be detected, and both maintenance and updates can be done in an effective way. The effort to apply these principles is considerable, so if the work done in the initial stages of development can be latter on used to guide, and validate the resulting system, it should. The current chapter proposes a formal development process, and illustrates the initial formal modeling of the requirements, and two levels of verification obtained through model checking and mathematical proof of correction.

The proposed development process starts with requirement analysis, where formal specifications are written to capture those requirements with a precise, unambiguous, and tractable notation. At this point the specification should include the key system's data types with the respective invariants, and some high level abstract functionalities. Because the requirements are captured in a formal language, it is possible to, already at this stage, reason about the integrity of the system. Depending on the chosen languages there are tools that can interpret the formal models, generate instances of the modeled system, gather integrity properties concerning the model, or even verify its mathematical correction. The verification effort should start almost alongside with the requirements analysis. It all depends on the capabilities of the chosen technologies, that should have some characteristics that suit the intended purpose. In this project the chosen technologies are: VDM++ for behavioral modeling, testing, and prototyping; Alloy for declarative modeling, and model checking; and HOL for mechanized proof discharge. Many other tools could apply such as Z, Communicating Sequential Processes (CSP) [40], and Coq, for example.

3.1 Abstract Modeling

Abstracting mean simplifying in order to focus on what's important, or overlooking some key aspect of an object or system in order to obtain a simplistic view. In order to take advantage through abstraction it is necessary to abstract just enough to clear all the redundant or insignificant details, although not so much that key aspects of the system do not get to be taken in account. This is the

main difficulty in building an abstract model. Some times, when dealing with systems of considerable size and complexity, it is better to produce different abstractions of the same system, that can relate to each other in a sound way, each capturing different properties and functionalities. In these cases, a clear architecture should be defined, relating the different concepts behind the models, so that the complete system is covered.

During this project, models in VDM++, Alloy, and HOL of the same abstract file system were written, and in the author's perspective there is much to gain, not only in writing different models for different parts of a system, but also for the same parts. It is particularly profitable to the understanding of underling properties, to express the same model in different modeling "paradigms", much alike observing the same phenomena from different scientific perspectives.

3.2 Model Translation

Translations are a key aspect of integrating tools and methods. The only actual mechanized translation of models available in the beginning of the project was that from VDM++ to HOL, using the Overture APS [92]. Aside from that the Alloy and VDM++ models are manually written in parallel, which is not bad for gaining insight and confidence in the model. However, having a VDM++ to Alloy translator would make the whole process of verification of VDM++ models amenable to VDM++ users that do not "speak" Alloy. Furthermore, a tool for translating Alloy to VDM++ would also be profitable to the Alloy community, that would gain the prototyping and automatic proof capabilities from the VDMTools and Overture respectively.

3.2.1 VDM++ to Alloy Translation: Hand Guide

In order to translate a VDM++ model to an Alloy model it is necessary to translate the data types, functions, operations, and proof obligations. If the operations are written just by articulation of functions, the translation of the operations is not necessary.

One thing to avoid is over restriction, because the model finding technique may result in unsuspected false positives. Alloy allows for model restriction using facts and multiplicity factors, much alike VDM invariants, although in this case the restriction automatically excludes invalid states from the search scope. The case is that, considering a file system specification, if a fact is inserted stating that all paths are well formed, it will not be possible to check if an operation yields a file system with malformed paths, because there will be no instance of the specification that can have such paths. This can lead to a situation where the Alloy Analyzer is asked to check an assertion and it will respond with no counter-examples found, although in fact the search was made on an empty set of examples, thus the inability to find any. To avoid this, it is possible to ask Alloy to show instances of the model, and if it does show any, then the set of examples is not empty. Another way to deal with over restriction is to declare restrictions as predicates instead of facts, or type multiplicity factors, and to selectively enforce them when checking assertions. In the way restrictions can be turned on or off for any checked property.

Some of the initial experiments with Alloy resulted in the development the *RelCalc* library (see Appendix A), based on the relational algebra lemmas to implement many utility predicates to constraint relations. Whenever useful the predicates are written in point free notation.

Types

VDM's type system semantics is quite complete, and allows for the declaration of explicit invariants regarding data types, which themselves can contain implicit invariants. In oder to translate the VDM explicit invariants, the corresponding logic expressions will be re-written in an Allow predicate, named after the original VDM data type's name, appended with the suffix *Invariant*. Two examples of data types with implicit invariants are the mapping and the non empty sequence.

Mapping — whenever a mapping is declared in a VDM model, it represents a binary relation from elements of a domain data type to elements of a range data type, that is simple and partially defined on the domain.

Non empty sequence — as the name says, empty sequence do not belong to this data type.

It is possible to introduce implicit invariants in the Alloy type signatures, although the option was not to do so. Instead, when translating a VDM data type, an Alloy predicate should be created stating the implicit constraints, if any. Another consequence resulting from the same choice, is that invariants on sub-types must be explicitly enforced. These extra type predicates will be named with the original VDM data type name, and the suffix *InvariantVDM*. Each *InvariantVDM* suffixed predicate also enforces the corresponding *Invariant* suffixed predicate.

Translating data types can be done in many ways, some of which will be presented next.

Record — can be translated using Alloy's type signature fields to represent the record fields.

```
Record ::
  a : A
  b : B
  c : C;
```

```
sig Record {
  a : A,
  b : B,
  c : C
}

pred RecordInvariantVDM[r : Record] {
  AInvariantVDM[r.a] and
  BInvariantVDM[r.b] and
  CInvariantVDM[r.c] and
  RecordInvariant[r]
}

pred RecordInvariant[r : Record] {}
```

Mapping — is a partial and simple relation, which means that its domain set is restricted, and that every element in the domain is mapped to at most one element of the range set. In Alloy this can be expressed through a binary relation[1] between two sets, which by definition in Alloy are bounded to a scope whenever are instantiated.

```
Mapping = map A to B;
```

```
sig Mapping {
  map : A -> B
}
```

Simplicity and partiality can be expressed through multiplicity factors of the relation.

```
sig Mapping' {
  map : A -> lone B
}
```

However, this would be constraining the model through facts. The preferred way to introduce the simplicity constraint in the respective *InvariantVDM* predicate, using the *RelCalc* `Simple` constraint.

```
pred MappingInvariantVDM[m : Mapping] {
  Simple[m.map,B]       and
  AInvariantVDM[dom[m.map]] and
  BInvariantVDM[rng[m.map]] and
  MappingInvariant[m]
}
```

Sequence — is a new feature of Alloy 4, which makes the translation from VDM almost direct.

```
Sequence = seq of A;
```

```
sig Sequence' { contents : seq A }
```

The length of all sequences are set with the `seq` keyword for all sequences in the model. Sequences in Alloy pack an extensive library providing the most common functionality, and many more. An alternative approach to sequence translation is to abstract it as an acyclic precedence relation between elements of the same type.

```
abstract sig Sequence {}

sig Element extends Sequence {
  tail : Sequence
}
sig Empty extends Sequence {}

pred SequenceInvariantVDM[s : Sequence] {
  Function[tail,Element,Sequence]      and
```

[1] In fact the `map` relation as defined here is a ternary relation from `Mapping` to `A` to `B`.

```
  Acyclic[(id[Element]).tail,Element] and
  SequenceInvariant[s]
}

pred SequenceInvariant[s : Sequence] {}
```

This way the declaration of a sequence is not made by instantiation of a parametric structure, and it is possible to set different scopes for each abstracted sequence.

Set — is a unary relation in Alloy, and can be expressed by signatures, or explicitly using the appropriate keyword.

```
Set = set of A;
```

```
sig Set { contents : set A }
```

Enumeration — can be achieved using the a combination of an `abstract` signature, denoting the VDM enumeration type, and extensions to that signature denoting the VDM enumerated values. There is a detail, that is not mandatory, although saves some memory in object instantiation, which consists in using the multiplicity factor `one` for the extension signatures. This assures that there will only be one instance of each enumerated value present in the model, without the need to adjust the scope.

```
Enumeration =
    <OneThing>
  | <AnotherThing>
  | <YetAnotherThing>;
```

```
abstract sig Enumeration {}

sig OneThing        extends Enumeration {}
sig AnotherThing    extends Enumeration {}
sig YetAnotherThing extends Enumeration {}
```

Optional — can be achieved similarly to the enumeration, by declaring an `abstract` signature, denoting the original VDM type, and two extensions to that signature. One for the VDM `nil` and another for the actual value.

```
Optional = [A];
```

```
abstract sig Optional {}

sig OptionalValue   extends Optional {}
one sig OptionalNil extends Optional {}
```

Numeric — data types other than integers do not have a match in Alloy. Thus floating point numbers available in VDM can only be abstracted as objects, and maybe some of the properties of those numbers can be modeled. Alloy integers have two forms, primitive integers and objects that carry the formers. The objectified integers can be atoms in relations, and quantified over, while the primitive integers can be combined and compared using arithmetic operators. The problem using integers is that to represent a number of a modest size such as 512 it is necessary to use a scope of 11, and for a very simple model, checking an assertion can take up to several minutes to perform on a regular laptop. This could be overcome using also the negative numbers, although it would not be a benefit in terms of elegance and simplicity. So whenever possible it is best to abstract numeric representation, and focus on the structural modulation.

Token — is really easy to translate, because a simple type signature, that can have the `one` multiplicity key word for memory savings if there is no need to distinguish token type values, with no fields or extensions whatsoever.

```
Token = token;
```

```
sig Token {}
```

Functions

VDM functions can have pre- and post-conditions, which the Alloy counterpart does not. These can be introduced as predicates with the prefixes *pre_* and *post_* followed by the name of the VDM function being translated.

```
suc : nat -> nat
suc(n) == n + 1
pre n > 0
post RESULT > n;
```

The translation can be done directly to Alloy function, by defining the input and output data types, together with an expression denoting the body of the function.

```
fun suc'[n : Int] : Int {
  n + 1
}

pred pre_suc[n : Int] {
  n > 0
}

pred post_suc[n,n': Int] {
  n' > n
}
```

Again, there is another possibility for the translation, which consists in abstracting the functions has relations and use predicates instead. With this approach it is necessary to distinguish the input

parameters from the output parameters, remember that Alloy predicates do not have a return type. The choice here is to follow a notational convention of Z syntax, where the output values of schemas have the prime character appended to the end of the name.

```
pred suc[n,n': Int] {
  n' = n + 1
}
```

Operations

Much in the same way as for VDM functions, VDM++ operations get translated to Alloy predicates, together with the respective pre- and post-conditions.

```
Sum : nat ==> nat
Sum(n) ==
  (s := s + n;
   return s);
```

```
pred Sum[state,state',n,n' : Int] {
  state' = state + n and
  n' = state
}
```

Proof Obligations

VDM++ POs are first order logic expressions, usually quantified over some set of variables. These logical expressions can be almost directly translated to Alloy assertions, with the same quantified variables. VDM++ input and output values to functions and operations are assumed to hold the respective invariants, otherwise there would be a run time exception. The same thing happens with quantified variables, sets, sequences, records, and mappings. However, in Alloy if there are facts declared in the model, then all values that could violate those facts are left out of the equation, without any warning whatsoever. Furthermore if we have avoided facts and have specified constraints as predicates, it is mandatory to explicitly enforce the constraints whenever necessary.

There can be some scope adjustments recommended for memory optimization when checking assertions. It is possible to enforce Alloy Analyzer to use just one instance of enumerated type values, optional type `nil` value, and sequence termination values.

The typical PO translation starts with the quantification followed by the enforcement of the invariants on the quantified variables implying the rest of the PO expression. Many examples of PO translations can be found in Chapter 4

3.2.2 VDM++ to HOL Translation: Preparation

To translate a VDM++ model and its POs to an HOL model, the APS provides the `VdmHolTranslator` that will automatically generate the HOL model. However, the `VdmHolTranslator` does not expect a VDM++ model as input to the translation process. What it does expect is a (1) VDM++

`OmlDocument`[2] (an OmlAst representation) of a VDM++ model, together with a (2) VDM++ value containing a set of instances of the `ProofObligation`[3] class. In order to automatically obtain (1) and (2) from the VDM++ model two parsers, a bash script and a VDM++ class have been built during this project. These tools were developed in a very ad hoc manner, mainly because the intention was simply to speed up the preparation of the models used in this project. However the design of the parsers can be greatly improved if built on proper grammars of their respective input, that in one case is the PO file (with the extension ".pog") produced by the VDMTools[4], and in the other is a textual representation of an `OmlDocument` object. The complete process of pre-processing VDM++ models in to (1) and (2), illustrated by Figure 3.1, will be presented next, together with the developed tools.

Step 1 — Type check, OmlAst and POs generation. Type checking and generation of the proof obligations is done by the VDMTools using the following commands:

```
vppde -t model.vpp
vppde -g model.vpp
```

Should the model pass the type checker with no errors[5], the POs can be generated to a file (`model.vpp.pog`). To generate the OmlAst representation of the VDM++ model the Overture Parser is invoked.

```
parser -pp model.vpp
```

The `parse` command is a bash script that invokes the Overture Parser.

Step 2 — Process the POs file. In order to construct `ProofObligation` objects out of the generated POs, it is necessary to supply both an OmlAst representation and a `Classification`[6] (definition of the type), of each PO in the file. First file `model.vpp.pog` must be processed to extract both the type and the expression, of every PO. This is done by the `pog_extractor` parser that simply reads each line at a time identifying the beginning and end of each PO. The parser is written in JFLEX [50], and the associated Java logic is responsible for identifying the classification and expression of each PO.

```
pog_extractor model.vpp.pog
```

This will produce two files, one (`model.vpp.pog.map`) that holds a mapping of PO `identifier` to `Classification`, and another (`model.vpp.values.vpp`) with new VDM++ class where boolean values are declared, one for each PO, and defined by the respective PO's expression.

[2] The `OmlDocument` class is defined in the OmlAst specification.

[3] The `ProofObligation` class is defined in the Automatic Proof System specification. To construct a `ProofObligation` instance it is necessary to provide a OmlAst of the proof obligation expression and its classification.

[4] To generate a ".pog" file using the VDMTools in the command line interface, it is necessary to pass the option "-g" that at the time of the writing of this thesis was not described in any of the tool's usage descriptions.

[5] If instructed to type check a model in command line, the VDMTools will first scan the model with its syntactic checker, and will only proceed if there are no syntactic errors.

[6] The `Classification` class is defined in the Automatic Proof System specification.

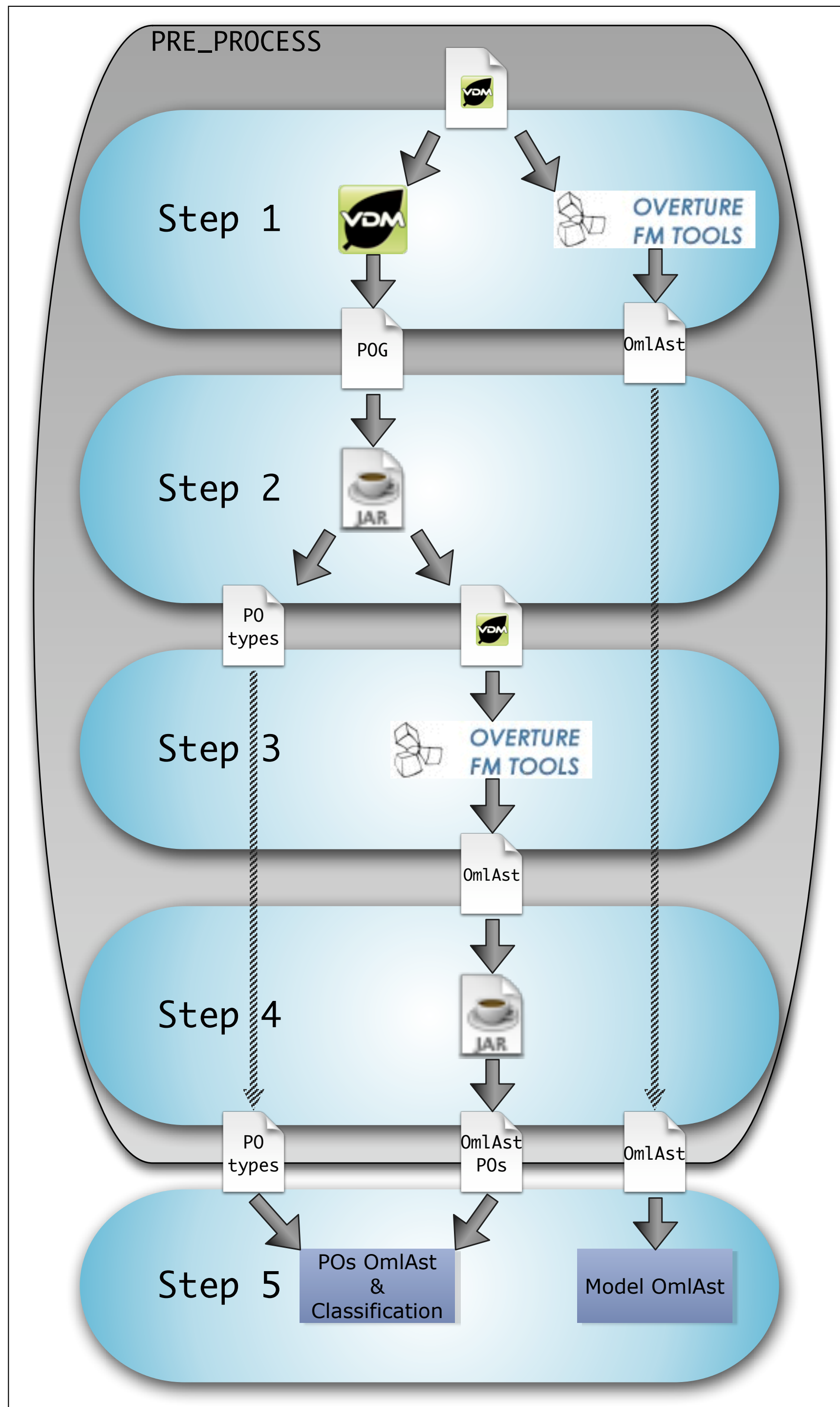

Figure 3.1: Preparation of VDM++ models for the VdmHolTranslator

The new auxiliary VDM++ class will enable the Overture Parser to generate a `OmlDocument` object containing all PO expressions as `OmlValueShape`[7] objects.

Step 3 — Generate OmlAst for each PO. At this point it is necessary to obtain a OmlAst representation for each PO, and this is done using the Overture Parser on the file generated in the previous step (`model.vpp.values.map`).

```
parse -pp model.vpp.values.vpp
```

This yields a new file (`model.vpp.values.vpp.app`) containing another `OmlDocument` object textual representation, from which the `OmlValueShape` objects still need to be extracted.

Step 4 — Extracting the OmlAst POs expressions. For this step another parser was developed, much alike the previous, that identifies every `OmlValueDefinition`[8] object, and extracts from it the respective `OmlValueShape` object. Again, this parser is not based on any established grammar for the input file, it simply parses it line by line and relies on observed properties of the `OmlDocument` objects issued by the Overture Parser.

```
pog_ast_extractor model.vpp.values.vpp.app
```

This will result in a new file (`model.vpp.vset`) containing a VDM set of `OmlValueShape` objects.

Step 5 — Generate a set of ProofObligation objects. The final step takes place inside the VDMTools execution environment, and it consists of merging the mapping of PO `identifier` to `Classification` value with the set of `OmlValueShape` objects to produce a set of instances of class `ProofObligation`. To achieve this a small VDM++ class, called `Extractor`, was written that iterates on each element of the set of `OmlValueShape` to extract their names and expressions, and builds a new set of `ProofObligation` objects, given that a corresponding `Classification` is found in the appropriate mapping. An `Extractor` object should be created within the VDMTools interpreter before invoking the `VdmHolTranslator`:

```
create extractor := new Extractor("model.vpp.vset", "model.vpp.pog.map")
print extractor.GetProofObligations()
```

The first four steps are integrated in the `pre_process` bash script, that provides the user the option of supplying a custom ".pog" file, so that it is possible to choose which POs are to be included in the final HOL model.

3.3 Verification

The development process explained in the former chapter, uses formal models in VDM++ and Alloy to annotate the system's requirements, test and model check the desired behavior. When the

[7]The `OmlValueShape` class is defined in the OmlAst specification.

[8]The `OmlValueDefinition` class is defined in the OmlAst specification.

system's models are written and validated from a functional point of view, is time to make use of the VDMTools' Integrity Checker to generate a set of POs, regarding the VDM++ model's integrity. The POs must now be discharged in order to validate the model in terms of it's integrity. The strategy is to use testing, model checking and mathematical proof of correction to verify the formal models, trying to uncover flaws using the "cheapest" method available, although stepping through all three methods when necessary.

The objective of the "multiple tool single PO" approach is to increase confidence in that the POs will hold, before an actual proof of correction is done. In many situations the stakeholders are only interested in increasing the bugs found, instead of completely proved correction. Verifying the model for *satisfiability* [45, 66], consists in: for every operation Op whose input is of type A and whose output is of type B, PO

$$\forall\ a\ \cdot\ a \in A \wedge \text{pre-}Op\ a \Rightarrow \exists\ \ \cdot b\ \cdot\ b \in B \wedge \text{post-}Op(b, a) \tag{3.1}$$

should be discharged, where $a \in A$ and $b \in B$ check for the invariants associated to A and B, respectively. Since all our operations are deterministic, the POs we have in hands are actually simpler:

$$\forall\ a\ \cdot\ a \in A \wedge \text{pre-}Op\ a \Rightarrow Op(a) \in B \tag{3.2}$$

The following situations can take place:

1. Op satisfies (3.2) although is semantically wrong — its does not behave according to the requirements. This calls for manual tests, which may include running the model as a prototype in an interpreter.

2. Op survives all tests compiled in the previous step (including dynamic type checking) and yet it does not satisfy (3.2) and the testers are not aware of this. In this case, a *model checker* able to automatically generate counter-examples to (3.2) which could suggest how to improve Op is welcome.

3. The model checker of the step just above does not find any counter examples. In this case a theorem prover is welcome to mechanically check (3.2).

4. PO (3.2) is too complex for the theorem prover we have available. In this situation, our ultimate hope is a pen-and-paper manual proof, or some kind of exercise able to decompose the too complex PO into smaller sub-proofs.

Satisfiability is evaluated for the entire system starting with testing, then model checking, and possibly ending with the proof of correction.

3.3.1 Testing

Testing for a specific PO does not make much sense when compared with the greater coverage a model checker can have, although because VDMTools as a dynamic checking system, whenever

a test suite is executed the dynamic checker looks for invariants, partial functions applications, mapping applications, and more violations. Any of these violations would most certainly result in a failed PO. Some specific tests can be written for each PO, although seems quite ineffective when compared with model checking.

3.3.2 Model Checking

The VDM++ generated POs are boolean values that can easily be written as an Alloy assertion, which can be checked with the Alloy Analyzer. Depending on the differences between the abstraction level of the VDM++ and Alloy models, some POs might not be representable in the Alloy model.

Alloy has another great advantage for this project, and it is the possibility of writing point free expressions based on relations. This connects perfectly well with the point free style of proofs, and the Alloy Analyzer can even be used to check conversions between point free and point wise definitions.

The Alloy Analyzer represents a given model by a boolean formula that is evaluated in an external SAT solver tool, that checks it for combinations of variables that evaluate it to false. If the SAT solver finds any of these combinations, Alloy Analyzer translates it back to an instance of the model to be inspected.

3.3.3 Proof of Correction

The ultimate degree of confidence given by this strategy of verification is the mathematical proof of correction, that can assure that for any given input the PO will hold. In the VDM community there is a long tradition of hand made proofs based on discrete maths to verify POs, although these take much time and require much more knowledge from the software engineer. Automated proofs can be done in HOL, directly from the VDM++ model, so that is the preferential option.

The translated HOL model is loaded in to HOL, together with the proof commands translated from the POs. There are two tactics for proof defined in the Overture APS, the `VDM_GENERIC_TAC` and the `VDM_ADDITIONAL_TAC`. The first tactic is mainly meant for domain-related POs, as the former is intended for the remaining proofs. Some times the tool does not choose the correct tactic, so whenever a PO is not discharged with one tactic, trying the other might solve the issue.

3.3.4 Tool Chain

The integration of methods and tools, for formal development and verification of software, proposed in this thesis is illustrated in Figure 3.2. From a given problem it is possible to build an executable specification in VDM++ that can rapidly be checked through unit testing, to validate if the ideas and concepts were modeled as expected. Using the VDMTools' Integrity Checker a set of proof obligations, denoted in VDM++ syntax, is automatically generated from the VDM++ specification. Provided an adequate translation of both VDM++ model and proof obligations to Alloy, the Alloy Analyzer can be used, again automatically, to model check the proof obligations. If it finds counter examples, then it means that the specification is not correct because there is an instance of the

Alloy model for which a proof obligation is invalid. Using Alloy Analyzer counter-examples it is easy to understand the conditions that lead to it, and go back to the VDM++ model to fix it. Otherwise, if the Alloy Analyzer does not find any counter-examples, the confidence in the integrity of the specification increases, although there is still room for unchecked design flaws. At this point many stakeholders would be more than satisfied with the gained confidence in the specification, however if a proof of correction is required it is possible to use the Overture APS to translate the VDM++ specification, together with the proof obligations, to HOL. If the theorem prover successfully discharges the proof obligations, the cycle terminates. On the other hand, if it fails, then there is a possibility that the proof can be simplified or even completed by hand.

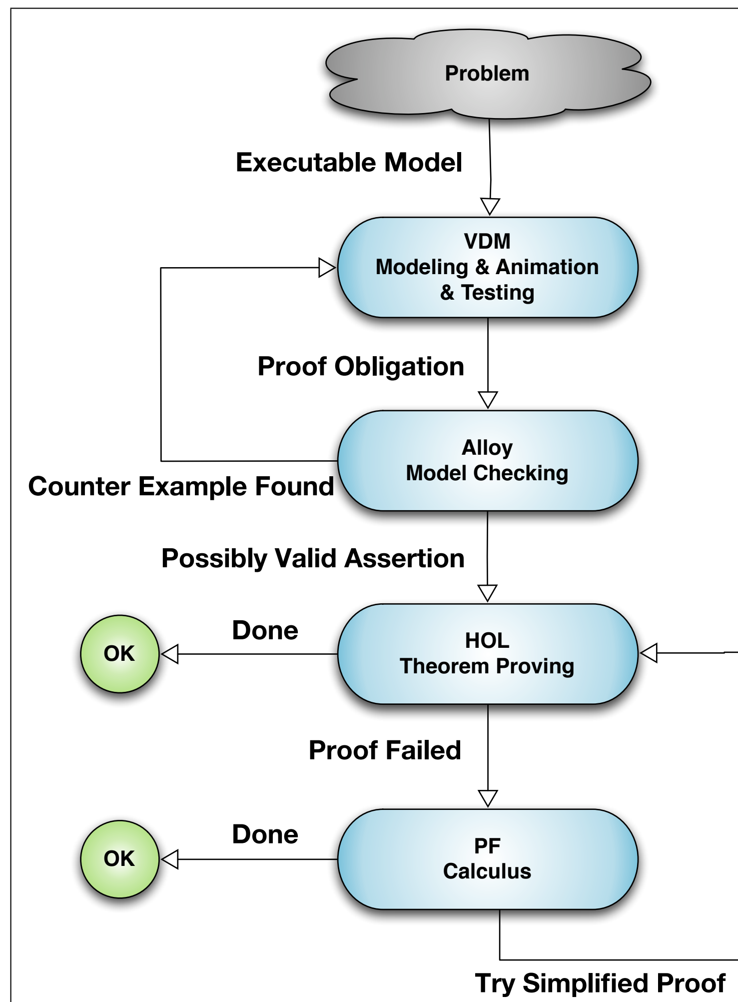

Figure 3.2: VDM — Alloy — HOL Tool Chain

The major drawback to the tool chain presented in Figure 3.2 is the fact that two specifications, VDM++ and Alloy, have to be hand written. Note that this is not necessarily a bad thing from the specification quality point of view, because using different modeling paradigms is in fact useful to get increased insight on the kernel of the problem at hand. However, it increases the required knowledge one must have to master both languages and tools, the probability of typos in the speci-

fications, and makes it harder to manage updates and fixes across multiple specifications.

3.4 Summary

The present chapter proposes a development process using formal specification amenable to verification from the beginning, in an effort to detect as many design flaws as possible in early stages.

Using VDM++ as a specification language to capture the initial requirements of a given object or system, it is easy to test, simulate and prototype in order to validate those requirements. However, to model check or mechanically discharge proof obligations other tools must be used, provided the adequate language translations.

Although automatic translation tools provide a sound integration between different languages, they are not always available. Section 3.2 provides translation rules for a VDM++ to Alloy conversion, and tools developed to speedup the preparation of VDM++ models to be used in the APS. These two translation steps are part of a verification tool chain intended to verify VDM++ models. Consistency of a VDM++ model can be established through discharging all POs generated from the model. In this sense a tool chain is designed to use multiple tools to deal with each PO.

The tool chain here presented is intended to be a contribution on its own to the GC initiative, whose mentors pointed tool integration as one of its main goals.

Chapter 4

Intel® Flash File System Core

The IFFSCRG document defines the architecture of a file system, some data structures, and the API of each layer of the architecture. Although the API is clear, there is almost no information about the internals of the file system, except for some sparse references to sequence tables.

Only the FSL was modeled, although studies have been made on how to model the complete architecture. The development started with an abstract version of the FSL, and can follow two possible paths: continue by writing abstract versions of the remaining layers; or the FSL model could be refined an decomposed in to the remaining layers. This is possible because the FSL is just the public API of the file system, and relies on the other layers to perform the actual management of the system's data.

Usually the formal modeling process starts by defining convenient data structures based on requirements analysis, although in the case of modeling an established API, which provides information on the operations of a given system, instead of its *modus operandi*, it is useful to model the system backwards, from the front-end operations back to the core data structures. This way becomes clear what are the data requirements for each operation, as details are introduced incrementally to support new ones. The intention is to model the file system internals in order to implement the API described in the document, trying to use as much information from the documentation as possible, whenever its relevancy so justifies.

For every operation described by the API there is a C alike function signature, identifying the operation's name, return value, and parameters. There is also a table describing the parameters, and stating which perform input (IN), output (OUT), or both (IN/OUT). The provided information is complete with a list of possible return values and a textual description of the operation. One common flaw throughout the document is that the return values regarding error status are not mapped to specific conditions of invocation nor file system state. So to use those status / error values it is necessary to assume a great portion of the operations behavior. Some times there are references to error situations in the parameters description, also in the operation description, although there is no association with specific status values.

Some of the most complex aspects of the system will be ignored, or this would not be an abstract enough model for structural verification purposes. POSIX defines different kinds of file in [82] (page 111 Section 3.163), although only regular files and directories will be modeled. Character special

File System API Reference intel®

4.6 FS_DeleteFileDir

Deletes a single file/directory from the media

Syntax

```
FFS_Status    FS_DeleteFileDir (
        mOS_char      *full_path,
        UINT8         static_info_type );
```

Parameters

Parameter	Description
*full_path	(IN) This is the full path of the filename for the file or directory to be deleted.
static_info_type	(IN) This tells whether this function is called to delete a file or a directory.

Figure 4.1: `FS_DeleteFileDir` operation (Permission to reproduce this excerpt is kindly granted by Intel Corporation).

file, block special file, First In First Out (FIFO) special file, symbolic links, and sockets, will be abstracted. Other things such as: open search handles, semaphores, ram buffers, concurrency will also be abstracted.

Two models have to be written for the FSL, one in VDM++ and another in Alloy. First the requirements will be denoted and tested in VDM++, and afterwards written and model checked in Alloy. The last step of the verification process will go through the translation of the VDM++ specification to HOL, and the discharge of POs in HOL. The complete specifications, test cases and assertions can be found in Appendix B, in addition to the complete reference of all project files in [53] (under the "Verified File Store" topic).

4.1 FS_DeleteFileDir

This operation deletes a single file or directory from the system, and is specified in page 30 of the IFFSCRG document. The signature and parameters description are in Figure 4.1

4.1.1 Requirements Analysis

The operation signature defines two input parameters `full_path` and `static_info_type`, although the later can be abstracted if it is possible to obtain that information by inspecting the type of the file pointer by the first parameter.

In textual description of the operation states that it *"deletes a single file or directory"*, along with all data associated with it. It also states some error situations:

- there should be an error "*if there are any open file handlers to the file specified*";
- there should be an error "*if there are any files or subdirectories in the directory specified*"

Additional information is provided regarding open search handles and that the operation will not return until it is finished, although, because open search handles and concurrency will be abstracted, these will not be part of the model. The focus of the formal modeling will be the requirements denoted next.

In order to delete a file one must supply a path, which means that files are referenced through paths.

R1 THERE SHOULD BE WAY TO STORE FILES, POINTED BY PATHS.

R2 THERE SHOULD BE WAY TO DELETE STORED FILES.

So far the document provided information on an important data type `FFS_Status`, that is defined in page 53, Section 4.21 Error Codes, as an enumeration of possible error / status values. Although the names of the codes are meaningful, there is no description associated. The names seem to follow a prefix convention, although there is no explicit reference to it, neither it is consistent throughout the document. The prefix *FFS_* seems to mean that the values associated with it are used across multiple layers of the architecture, as *FS_* seems to mean that those values associated with it are used only at the FSL level. In the same way the *DO_* could be for status values used within the DOL. There is another group of values with the prefix *SEQ_*, however these values are not used any where in the document.

R3 THERE SHOULD BE WAY TO EXPRESS THE STATUS RETURN VALUES FOR THE OPERATION.

The second error case, obtained from the description, reveals a fundamental concept of file systems, that directories have sub-files, which themselves can be regular files or directories. It also reveals a constraint to the operation, that is the impossibility to remove directories that contain sub-files.

R4 FILES CAN BE REGULAR FILES OF DIRECTORIES.

R5 DIRECTORIES HAVE SUB-FILES.

R6 IT IS NOT POSSIBLE TO DELETE DIRECTORIES THAT HAVE SUB-FILES.

One other thing that is made quite clear is the fact that open files can not be deleted, and in those cases there should be an error. From this results a new requirement, that is to be able to keep track of open files, which is also quite obvious to whom might ever have used a file system. Open files are identified by *open file handles*, there is a definition for `FS_FileHandle` in page 260. A `FS_FileHandle` is a pointer for a structure `FS_OpenFileInfo`, defined on page 258 Section 15.2.7 FS_OpenFileInfo, which is intended to cache *"information about an open file"*.

R7 THERE SHOULD BE WAY TO KEEP TRACK OF OPEN FILES.

R8 IT IS NOT POSSIBLE TO DELETE OPEN FILES.

Still in the textual description of the operation reference is made to a "*dynamic info, primary data, or auxiliary data*" as child objects associated files and directories, from which the primary data and the auxiliary data is easy to understand, on the contrary dynamic info is a much vaguer concept. From the context it is assumed that dynamic info is what the file system stores on open files in main memory, and that static info would be stored in the FLASH memory. Again, this is just an assumption, because other assumptions could be made. There are references to dynamic data as data that can be modified and written using the device **mlc!** (**mlc!**) [2] capabilities or in Pseudo Single Bit per Cell (PSBC) [72].

R9 FILES HAVE DATA.

4.1.2 VDM++ Model

The requirements analysis found nine requirements for the `FS_DeleteFileDir` operation. Five of them are related to the internal data structures of the FSL, namely **R1**, **R4**, **R5**, **R7**, and **R9**; other three are related with the operation behavior, **R2**. **R6**, and **R8**; and a last one **R3** about status information returned by the operation (and all others for that matter).

The file system will manage files in general, although a file can be a regular file or a directory. Directories can have sub-files, forming a tree shape. This would suggest for a recursive tree such as structure, in fact many of the most common file systems are implemented that way. However taking in account, the fact that the specification will have to be verified, it might be an advantage no to use recursive structures, thus not having to perform inductive proofs. The nested structure of directories and sub-directories can be represented through a path from the top most directory (the root of the file system) to the file it references. This way the file store could be specified as a simple mapping from `Path` to `File`, provided an adequate invariant.

```
FileStore = map Path to File
inv fileStore ==
  forall path in set dom fileStore &
    let parent = dirName(path) in
    parent in set dom fileStore         and
    isDirectory(fileStore(parent).info);
```

Code 4.1: VDM++ model sliced at FS_DeleteFileDir — FileStore data type

The invariant of the `FileStore` data type states that all files have one parent file that is a directory, see [82] page 69 Section 3.263. POSIX also defines `dirname`, in [84] page 256, as a function that when given a path, returns the path of its containing directory (the parent file's path). In the case of the root directory, the parent may be the root itself, or simply not defined. When dealing with symbolic links POSIX is clear to state that the system calls will have to result in the error value `[ELOOP]` [84] page 24, if there are loops introduced by symbolic links in a given path. With out symbolic links and only dealing with absolute paths, it is not possible to have loops in

paths. Hence, in mathematical therms, `dirname`, as defined here, is reflexive for the root directory, and acyclic for all other paths.

```
dirName : Path -> Path
dirName(full_path) ==
  cases full_path:
    <Root> -> <Root>,
    [-]    -> <Root>,
    others -> [ full_path(i) | i in set inds full_path & i < len full_path ]
  end;
```

Code 4.2: VDM++ model sliced at FS_DeleteFileDir — dirName function

The function `dirName` assures that the parent of the root directory is always itself, as for paths not having cycles, it is assured by the non recursive nature of `FileStore`.

```
Path = <Root> | seq1 of FileName;
```

Code 4.3: VDM++ model sliced at FS_DeleteFileDir — Path data type

In addition to the requirements gathered from the documentation, two new ones are introduced.

AR1 THERE MUST BE A UNIQUE ROOT DIRECTORY.

AR2 THE ROOT DIRECTORY IS ITS OWN PARENT DIRECTORY.

About files it is specified that there can be two types: regular files, and directories. That directories can have sub files, and that regular files have data contents.

```
File :: info : FS_FileDirInfo;
```

Code 4.4: VDM++ model sliced at FS_DeleteFileDir — File data type

The `info` field stores information about using the structure `FS_FileDIrInfo`, defined in page 257. Only file's attributes are specified, because the remaining fields can be abstracted, as explained in Table4.1. The IFFSCRG document refers to auxiliary data contents of files, but also states that they are optional [1], so all these will be abstracted.

```
FS_FileDirInfo :: attributes : Attributes;
```

Code 4.5: VDM++ model sliced at FS_DeleteFileDir — FS_FileDirInfo data type

```
Attributes :: fileType : FileType;
```

Code 4.6: VDM++ model sliced at FS_DeleteFileDir — Attributes data type

```
FileType = <RegularFile> | <Directory>;
```

Code 4.7: VDM++ model sliced at FS_DeleteFileDir — FileType data type

[1] See the use of `FFS_AUXILIARY_FILE_DATA_ENABLED` in page 157, Section 15.2.6 FS_FileDirInfo.

Both `Attributes` and `FS_FileDirInfo` are records so that they can easily be expanded when necessary. The `FileType` data type has the same properties, although because it is simply a flag, it is defined as an enumerated type allowing for more kinds of files to be specified.

The fact that a file is open could be recorded in the `File` data type, although it is considered that information about open files should be maintained in main memory, in opposition to the actual file that is stored in the flash memory. Following this principle, there is a clear separation of the information about open files in a new data structure, where the information is referenced by a handler.

```
OpenFilesTable = map FS_FileHandle to FS_OpenFileInfo;
```

Code 4.8: VDM++ model sliced at FS_DeleteFileDir — OpenFilesTable data type

As defined in page 260 of the IFFSCRG document, file handlers are pointers to the information on open files.

```
FS_FileHandle = nat;
```

Code 4.9: VDM++ model sliced at FS_DeleteFileDir — FS_FileHandle data type

The information on open files is captured by the structure `FS_OpenFileDir`, defined in page 258. Being abstraction a priority of the formal modeling done here, `FS_FileHandle` would be a perfect candidate to be excluded from the specification, however the option here is to model it as a unique[2] numerical identifier. The fact that file handles, or descriptors, are the external reference to open files, which applications use to access their files, is the main reason behind the option not to abstract them.

```
FS_OpenFileInfo :: path : Path;
```

Code 4.10: VDM++ model sliced at FS_DeleteFileDir — FS_OpenFileInfo data type

[2]The uniqueness of the `FS_FileHandle` is not a consequence of any invariant related to the data type. It is derived from the simplicity of the mapping `OpenFilesTable`.

FileOffset	Abstracted because the model manages open files through the `OpenFilesTable`, it would only have to be persistent if the system should set the offset of newly open files to the last known value. For the present operation there is no requirement on open file's offset.
DataSize	Abstracted because contents size can be calculated on demand through the `len` function.
AuxDataSize	Abstracted because contents size can be calculated on demand through the `len` function. Auxiliary data is optional.
CreationTime	Time stamps are abstracted because they are not relevant to the operation.
LastWriteTime	Time stamps are abstracted because they are not relevant to the operation.
LastAccessTime	Time stamps are abstracted because they are not relevant to the operation.
*Filename	Abstracted because it is redundant as the file's full path is stored in the file store.

Table 4.1: Abstracted fields from structure `FS_FileDirInfo`

This structure defines multiple fields of information, and all of them can be abstracted, as shown in Table 4.2.

Field	Reason
Next	Abstracted because the model manages open files through the `OpenFilesTable`.
StaticInfoLoc	Abstracted because BAL Logical Unit is also abstracted.
ParentStaticInfoLoc	Abstracted because BAL Logical Unit is also abstracted. The parent's path can always be obtained through the `dirName` function.
PrimaryDataLoc	Abstracted because BAL Logical Unit is also abstracted. The data of an open file can be obtained using the file's path through the `FileStore`.
AuxDataLoc	Abstracted because BAL Logical Unit is also abstracted. The data of an open file can be obtained using the file's path through the `FileStore`. Auxiliary data is optional.
DataSize	Size counters can be abstracted by size counting functions.
FileOffset	Abstracted because it is only used by the operations for writing, reading and offset shift.
AuxDataSize	Size counters can be abstracted by size counting functions. Auxiliary data is optional.
AuxDataOffset	Abstracted because it is only used by the operations for writing, reading and offset shift. Auxiliary data is optional.
OldDataSize	Abstracted because transactions are also abstracted.
OldFileOffset	Abstracted because transactions are also abstracted.
PhysicalDataSize	Abstracted because transactions are also abstracted.
ShareMode	Abstracted because transactions are also abstracted.
AccessMode	Abstracted because it is not relevant for this operation. Open files can not be deleted.
RamBufferMode	Abstracted because ram buffers are also abstracted.
Modified	Abstracted because it is not relevant for this operation. The only changes produced by this operation are complete removal of files.
TransOpStatus	Abstracted because transactions are also abstracted.
SizeChanged	Abstracted because it is not relevant for this operation. The only changes produced by this operation are complete removal of files.
DeleteStatusMode	Abstracted because ram buffers are also abstracted.
PrimaryRootReplaced	Abstracted because transactions are also abstracted.

Table 4.2: Abstracted fields from structure `FS_OpenFileDir`

The only filed in `FS_OpenFileDir` is the path to the open file. This is one option when correlating `FileStore` with `OpenFilesTable`. Another way would be to have an indirection step, using an extra data structure. The correlation between these two data structures will lead to an referential integrity invariant, that will be discussed later on. So far the specification accounts for the requirements: **R1**, **R4**, **R5**, **R7**, and **R9**; and the two new ones **AR1** and **AR2**.

To meet requirement **R3** regarding the return status values, it is necessary to introduce a new

data type that enumerates the possible values. Not all will be considered, because some are not meaningful enough so that its use can be assumed, and others because the underlying concepts, such as file system initialization, concurrency, hardware errors, and memory limits, are being abstracted.

```
FFS_Status =
    <FFS_StatusSuccess>
  | <FS_ErrorFileNotFound>
  | <FS_ErrorFileStillOpen>
  | <FS_ErrorDirectoryNonEmpty>
  | <FFS_StatusUnknown>;
```

Code 4.11: VDM++ model sliced at FS_DeleteFileDir — FFS_Status data type

The system can now be modeled as a record containing the file store and the open files table.

```
System :: table     : OpenFilesTable
          fileStore : FileStore
inv sys ==
  forall ofi in set rng sys.table &
    isElemFileStore(ofi.path, sys.fileStore);
```

Code 4.12: VDM++ model sliced at FS_DeleteFileDir — System data type

The above mentioned referential integrity invariant, can only be expressed at the `System` level, where both structures are available. It states that all open files present in the table, must exists as files in the file store. The remaining requirements specify what the operation should **R2**, and should not **R6 R8** do. The `FS_DeleteFileDir` shall remove a file from the system's file store.

```
FS_DeleteFileDir_FileStore: FileStore * set of Path -> FileStore
FS_DeleteFileDir_FileStore(fileStore, paths) ==
  paths <-: fileStore
pre forall path in set dom fileStore &
    dirName(path) in set paths => path in set paths;
```

Code 4.13: VDM++ model sliced at FS_DeleteFileDir — FS_DeleteFileDir_FileStore function

From **R8** follows that the operation does not need to modify the open files table, hence no function is needed to manipulate the `OpenFilesTable` data type. At the `System` level there is no need to coordinate actions taken on the two sub-types, because only `FileStore` will play a role in this operation.

```
FS_DeleteFileDir_System: System * Path -> System
FS_DeleteFileDir_System(sys, full_path) ==
  mu(sys, fileStore |-> FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}))
pre (forall ofi in set rng sys.table & ofi.path <> full_path) and
    pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path});
```

Code 4.14: VDM++ model sliced at FS_DeleteFileDir — FS_DeleteFileDir_System function

The `FS_DeleteFileDir_System` function just enforces the pre-condition of its `FileStore` counter part. Above the `System` level the main function is total, and accounts for the error situations, relaying the selection of the appropriate status code to the exception handler function.

```
FS_DeleteFileDir_Main: System * Path -> System * FFS_Status
FS_DeleteFileDir_Main(sys, full_path) ==
  if full_path in set dom sys.fileStore and
     pre_FS_DeleteFileDir_System(sys, full_path)
  then mk_(FS_DeleteFileDir_System(sys, full_path), <FFS_StatusSuccess>)
  else mk_(sys, FS_DeleteFileDir_Exception(sys, full_path));
```

Code 4.15: VDM++ model sliced at FS_DeleteFileDir — FS_DeleteFileDir_Main function

The main function of an operation will always check the pre conditions of the underlying functions, so that their application is correct, although not all conditions to the invocation will be stated in pre-conditions. Pre-conditions will only record the predicates that enable a correct behavior of a function, at a specific data type level, although when composing the cascade of functions some other predicates might need to be checked at the main function, before the operation continues. This will not be the case with the `FS_DeleteFileDir` operation, although will happen with other operations.

```
FS_DeleteFileDir_Exception: System * Path -> FFS_Status
FS_DeleteFileDir_Exception(sys, full_path) ==
  if not isElemFileStore(full_path, sys.fileStore)
  then <FS_ErrorFileNotFound>
  elseif isElemTablePath(full_path, sys.table)
  then <FS_ErrorFileStillOpen>
  elseif isDirectory(sys.fileStore(full_path).info) and
         hasSubFiles(sys.fileStore, full_path)
  then <FS_ErrorDirectoryNonEmpty>
  else <FFS_StatusUnknown>;
```

Code 4.16: VDM++ model sliced at FS_DeleteFileDir — FS_DeleteFileDir_Exception function

The operation specification is complete with the error handling function, which selects the appropriate status code to be returned, and its objectified method `FS_DeleteFileDir`.

```
FS_DeleteFileDir : Path ==> FFS_Status
FS_DeleteFileDir(full_path) ==
  def mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path) in
  (sys := sys'; return status);
```

Code 4.17: VDM++ model sliced at FS_OpenFileDir — FS_DeleteFileDir operation

Testing starts right now, by creating test values, of all defined types, so that unit test can be performed. Just by writing the test values, and loading model in the VDMTools' interpreter the dynamic type checker automatically detects flaws in those values. In many cases this reveals flaws in the data type's definitions. The unit testing is done for both data types and operations. The data type tests follow the white box philosophy, by invoking the functions that manipulate the respective type and asserting the results. When testing the operations at top level, black box philosophy, the test cases invoke the outmost main function of the operation. This way, functions are tested out side of the context they were designed to work in, as well as part of the cascade that models an operation. The unit test experience shows that many typos get caught at this stage, situations where the modeler wants to express some thing although ends up writing another. Unit testing in

the presence of a dynamic type checker is a very powerful tool, that can save much time in further verification.

4.1.3 Unit Testing the VDM++ model

Testing the `FS_DeleteFileDir` operation is done following both the white box philosophy for the involved data types: `System` and `FileStore`; and the black box philosophy at the API level. Test cases are written for each and every level. Tests done at the data type levels will be asserting properties regarding the respective data type. Test done at the API level will cover all possible status values, as well as other possible return values.

The operation at hand removes files from the system's file store, hence the resulting file store shall not have the removed file, although all the other files should remain intact.

```
SimpleDeleteTest: set of Path ==> ()
SimpleDeleteTest(paths) ==
  let fs' = FS_DeleteFileDir_FileStore(fs,paths) in
  (AssertTrue(paths inter dom fs' = {});
   AssertTrue(paths union dom fs' = dom fs);
   fs := fs');
```

Code 4.18: Tests for VDM++ model sliced at FS_DeleteFileDir — SimpleDeleteTest at FileStore level

```
RunTest: () ==> ()
RunTest() ==
  (dcl init : FileStore := fs;
   SimpleDeleteTest({["etc", "hosts"]});
   SimpleDeleteTest({["etc", "conf.d"]});
   SimpleDeleteTest({["bin", "ls"]});
   SimpleDeleteTest({<Root>, ["etc"], ["etc", "resolv.conf"], ["bin"], ["bin","wc"]});
  );
```

Code 4.19: Tests for VDM++ model sliced at FS_DeleteFileDir — Test case at FileStore level

At the `System` data type level, the operation calls the respective `FileStore` function, updates the `fileStore` field with the result, and the `table` field is to be left just as it was.

```
SimpleDeleteTest: Path ==> ()
SimpleDeleteTest(path) ==
  let sys' = FS_DeleteFileDir_System(sys,path) in
  (AssertTrue(path not in set dom sys'.fileStore);
   AssertTrue(dom sys.fileStore = dom sys'.fileStore union {path});
   AssertTrue(sys.table = sys'.table);
   sys := sys');
```

Code 4.20: Tests for VDM++ model sliced at FS_DeleteFileDir — SimpleDeleteTest at System level

```
RunTest: () ==> ()
RunTest() ==
  (SimpleDeleteTest(["etc","hosts"]);
   SimpleDeleteTest(["etc","conf.d"]);
   SimpleDeleteTest(["etc","resolv.conf"]);
   SimpleDeleteTest(["bin","ls"]);
   SimpleDeleteTest(["bin","wc"]);
```

```
    SimpleDeleteTest(["etc"]);
    SimpleDeleteTest(["bin"]);
    SimpleDeleteTest(<Root>);
  );
```

Code 4.21: Tests for VDM++ model sliced at FS_DeleteFileDir — Test case at System level

The test coverage is almost 100% for the four functions, as shown in Table 4.4, however no test has triggered the `FFS_StatusUnknown` return value. This is according to the expected because the intention was to specify every predictable error situation with an expressive status value. This property will be checked in the next model checking step, to find out whether there is a combination of input parameters capable of triggering the unknown status value.

Name	**#Calls**	**Coverage**
FileSystemLayerOperations‘FS-DeleteFileDir-Main	4	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-System	9	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-FileStore	13	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-Exception	3	96%
Total Coverage		**98%**

Table 4.4: Test coverage for functional specification of `FS_DeleteFileDir`

At the API level the objectified operation is tested, and the assertions are focused on the possible return values. Test coverage is presented in Table 4.6.

```
RunTest: () ==> ()
RunTest() ==
  (dcl status : FFS_Status;
    status := FS_DeleteFileDir(["etc","hosts"]);
    AssertTrue(status = <FFS_StatusSuccess>);

    status := FS_DeleteFileDir(["bin","cp"]);
    AssertTrue(status = <FS_ErrorFileNotFound>);

    status := FS_DeleteFileDir(["etc"]);
    AssertTrue(status = <FS_ErrorFileStillOpen>);

    status := FS_DeleteFileDir(["bin"]);
    AssertTrue(status = <FS_ErrorDirectoryNonEmpty>);
  );
```

Code 4.22: Tests for VDM++ model sliced at FS_DeleteFileDir — Test case at API level

Name	**#Calls**	**Coverage**
FileSystemLayerObj‘FS-DeleteFileDir	4	$\surd$
Total Coverage		**100%**

Table 4.6: Test coverage for objectified specification of `FS_DeleteFileDir`

4.1.4 Alloy Model

Building a semantically equivalent Alloy model of the VDM++ model, will follow the directions pointed in Section 3.2.1. Whenever more than one possible translation can be used, the option will be to abstract as much as possible.

`FileStore` is translated to a signature that contains a `map` field, which is a relation from `Path` to `File`[3]. The `map` relation must be constrained, through the `FileStoreInvariantVDM` predicate, so that it has the same properties as the VDM++ `FileStore` mapping. How to build the *InvariantVDM* predicates was also explained in Section 3.2.1.

[3] In fact the `map` relation defined in the `FileStore` signature is a ternary relation from `FileStore` to `Path` to `File`.

```
sig FileStore {
  map: Path -> File
}
```

Code 4.23: Alloy model sliced at FS_DeleteFileDir — FileStore signature

```
pred FileStoreInvariantVDM[fs: FileStore]{
  RelCalc/Simple[fs.map, File]           and
  PathInvariantVDM[RelCalc/dom[fs.map]]  and
  FileInvariantVDM[RelCalc/rng[fs.map]]  and
  FileStoreInvariant[fs]
}
```

Code 4.24: Alloy model sliced at FS_DeleteFileDir — FileStoreInvariantVDM predicate

Initially the `FileStoreInvariantVDM` predicate had an extra clause where it was stated that the `map` relation had to be injective, and this fact was not derived from the semantics of the VDM++ mappings, nor from the fact that sub-type invariants must be explicitly enforced. It was based on the fact that VDM records, in this case `File`, are created using the `mk_` constructor, that for every invocation always creates a new value of the desired type. In other words, even if files are created with the same parameters, in the VDM++ model, they are still different files. Another thing that supported this constraint is the fact that, for now, links are abstracted, and if two different paths in the `map` relation would reference the same file, could lead to the interpretation that there would be a link for the file. However, this would be adding a conceptual constraint related to an interpretation issue, hence not absolutely necessary. For abstraction sake, the `map` relation does not have to be injective, although whenever to paths map in the same file that should be interpreted as two different files that are equal. In the next operation the case will be different, and the injectiveness of the mapping relation will not be just a question of interpretation.

```
pred FileStoreInvariant[fs: FileStore]{
  (fs.map).(File->Directory) in dirName.(fs.map).info.attributes.fileType
}
```

Code 4.25: Alloy model sliced at FS_DeleteFileDir — FileStoreInvariant predicate

The actual `FileStore` invariant is written using the point free transform, where all variables were removed and the predicate is purely relational. This is not easily done within an automatic translator, although still it illustrates well the relational expressiveness of Alloy. Paths in the VDM++ model, are disjoint unions of the type `<Root>` and non empty sequence of `FileName`. There is also a function `dirName` that computes the path to the parent directory of a given path. In the Alloy model the `Path` signature is more abstract, because it does not state that paths are constructed with sequences, and that there is a function that computes the parent's path.

```
abstract sig Path {
  dirName: Path
}
```

Code 4.26: Alloy model sliced at FS_DeleteFileDir — Path signature

Paths are modeled as an abstract signature containing a field named `dirName`, that relates a path to its parent path. It is necessary to extend the `Path` signature, to express the disjoint union.

```
one sig Root extends Path {}
```

Code 4.27: Alloy model sliced at FS_DeleteFileDir — Root signature

```
sig FileNames extends Path {}
```

Code 4.28: Alloy model sliced at FS_DeleteFileDir — FileNames signature

The multiplicity factor `one` assigned to the `Root` signature means that there will be only one value instantiated to this signature. This could be done in the *InvariantVDM* predicate, although this way the objects scope for every command will automatically be adjusted. The `PathInvariantVDM` predicate must assure that the relation `dirName` will show equivalent properties to the VDM++ function that has the same designation.

```
pred PathInvariantVDM[path : Path]{
  dirNameProperties
}
```

Code 4.29: Alloy model sliced at FS_DeleteFileDir — PathInvariantVDM predicate

The predicate `dirNameProperties` states that the relation is in fact a function, that it is reflexive on the root directory, and that it is acyclic for every other path.

```
pred dirNameProperties {
  RelCalc/Function[dirName,Path,Path]                 and
  RelCalc/Reflexive[(RelCalc/id[Root]).dirName,Root]  and
  RelCalc/Acyclic[(RelCalc/id[FileNames]).dirName,FileNames]
}
```

Code 4.30: Alloy model sliced at FS_DeleteFileDir — dirNameProperties predicate

Files in the Alloy model are signatures containing only the information field. The `FileInvariantVDM` predicate enforces the sub-type invariant of `FS_FileDirInfo`.

```
pred FileInvariantVDM[f: File]{
  FS_FileDirInfoInvariantVDM[f.info]
}
```

Code 4.31: Alloy model sliced at FS_DeleteFileDir — FileInvariantVDM predicate

The `FS_FileDirInfo` and `Attributes` are records in the VDM++ model, and translated in the same way as `File`.

```
sig FS_FileDirInfo {
  attributes      : Attributes
}
```

Code 4.32: Alloy model sliced at FS_DeleteFileDir — FS_FileDirInfo signature

```
pred FS_FileDirInfoInvariantVDM[info: FS_FileDirInfo] {
  AttributesInvariantVDM[info.attributes]
}
```

Code 4.33: Alloy model sliced at FS_DeleteFileDir — FS_FileDirInfoInvariantVDM predicate

```
sig Attributes {
  fileType: FileType
}
```

Code 4.34: Alloy model sliced at FS_DeleteFileDir — Attributes signature

```
pred AttributesInvariantVDM[attr: Attributes] {
  FileTypeInvariantVDM[attr.fileType]
}
```

Code 4.35: Alloy model sliced at FS_DeleteFileDir — AttributesInvariantVDM predicate

Enumerations get translated as an abstract signature, that is extended by the different possible values.

```
abstract sig FileType {}
```

Code 4.36: Alloy model sliced at FS_DeleteFileDir — FileType signature

```
one sig RegularFile extends FileType {}
```

Code 4.37: Alloy model sliced at FS_DeleteFileDir — RegularFile signature

```
one sig Directory extends FileType {}
```

Code 4.38: Alloy model sliced at FS_DeleteFileDir — Directory signature

Just such as with the `Path` signature, the option of including the `one` multiplicity factor in the above signatures is simply to save resources in model finding. The `OpenFilesTable` mapping, and the `FS_OpenFileInfo` record, are translated to Alloy in the same way as the `FileStore`, and `File` respectively.

```
sig OpenFilesTable {
  map: FS_FileHandle -> FS_OpenFileInfo,
}
```

Code 4.39: Alloy model sliced at FS_DeleteFileDir — OpenFilesTable signature

```
pred OpenFilesTableInvariantVDM[table: OpenFilesTable] {
  RelCalc/Simple[table.map,FS_OpenFileInfo]           and
  FS_OpenFileInfoInvariantVDM[RelCalc/rng[table.map]]
}
```

Code 4.40: Alloy model sliced at FS_DeleteFileDir — OpenFilesTableInvariantVDM predicate

```
sig FS_OpenFileInfo {
  path      :  Path
}
```

Code 4.41: Alloy model sliced at FS_DeleteFileDir — FS_OpenFileInfo signature

```
pred FS_OpenFileInfoInvariantVDM[ofi: FS_OpenFileInfo]{
  PathInvariantVDM[ofi.path]
}
```

Code 4.42: Alloy model sliced at FS_DeleteFileDir — FS_OpenFileInfoInvariantVDM predicate

Because `FS_FileHandle` is the external reference for open files, and it will not be used for any arithmetic, there is no need to specify it as an Alloy integer.

```
sig FS_FileHandle {}
```

Code 4.43: Alloy model sliced at FS_DeleteFileDir — FS_FileHandle signature

The operation at the different levels is translated to the next six predicates.

```
pred FS_DeleteFileDir_Main[sys, sys': System, full_path:Path, status: FFS_Status] {
  (full_path in RelCalc/dom[sys.fileStore.map] and
   pre_FS_DeleteFileDir_System[sys,full_path])
  =>   (FS_DeleteFileDir_System[sys,sys',full_path] and
        status = FFS_StatusSuccess)
  else (FS_DeleteFileDir_Exception[sys,full_path,status] and
        sys' = sys)
}
```

Code 4.44: Alloy model sliced at FS_DeleteFileDir — FS_DeleteFileDir_Main predicate

```
pred FS_DeleteFileDir_System[sys, sys': System, full_path: Path] {
  FS_DeleteFileDir_FileStore[sys.fileStore,sys'.fileStore,{full_path}] and
  sys.table = sys'.table
}
```

Code 4.45: Alloy model sliced at FS_DeleteFileDir — FS_DeleteFileDir_System predicate

```
pred pre_FS_DeleteFileDir_System[sys: System, full_path: Path] {
  full_path not in RelCalc/rng[sys.table.map].path        and
  pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
}
```

Code 4.46: Alloy model sliced at FS_DeleteFileDir — pre_FS_DeleteFileDir_System predicate

```
pred FS_DeleteFileDir_FileStore[fs,fs': FileStore, paths: set Path] {
  fs'.map = fs.map - (paths -> (paths.(fs.map)))
}
```

Code 4.47: Alloy model sliced at FS_DeleteFileDir — FS_DeleteFileDir_FileStore predicate

```
pred pre_FS_DeleteFileDir_FileStore[fs: FileStore, paths: set Path] {
  (((Path - paths)->Path) & iden).(fs.map) in dirName.((Path - paths)->File)
}
```

Code 4.48: Alloy model sliced at FS_DeleteFileDir — pre_FS_DeleteFileDir_FileStore predicate

```
pred FS_DeleteFileDir_Exception[sys: System, full_path: Path, status: FFS_Status]{
  not isElemFileStore[full_path,sys.fileStore]
  =>   status = FS_ErrorFileNotFound
  else (isElemTablePath[full_path,sys.table]
       =>   status = FS_ErrorFileStillOpen
       else ((isDirectory[(sys.fileStore.map[full_path]).info] and
              hasSubFiles[sys.fileStore,full_path])
             =>   status = FS_ErrorDirectoryNonEmpty
             else status = FFS_StatusUnknown))
}
```

Code 4.49: Alloy model sliced at FS_DeleteFileDir — FS_DeleteFileDir_Exception predicate

4.1.5 Model Checking the Operation with the Alloy Analyzer

Model checking usually reveals design flaws, where the specification expresses exactly what the specifier intended, although it still allows for unpredicted behavior. These are usually the most expensive to solve if not detected at an early stage of development. Operations are specified in Alloy as a cascade of predicates, where there are predicates for each relevant type signature, a *_Main* predicate that is the starting point of the cascade, and an error handling *_Exception* predicate. To model check an operation, assertions are made about each separate predicate that takes part in the cascade. The assertions made about the *_Main* predicate will check the success and error cases generally, as the assertions made about the *_Exception* will extensively check the different error situations. The type signature related assertions will check the respective predicate for satisfiability [66].

The `FS_DeleteFileDir_FileStore` predicate is checked for satisfiability by the following pair of `assert` and `check` commands.

```
assert Delete_FileStore {
  all fs,fs': FileStore, paths: set Path {
    FileStoreInvariantVDM[fs]                 and
    PathInvariantVDM[paths]                   and
    pre_FS_DeleteFileDir_FileStore[fs,paths]  and
    FS_DeleteFileDir_FileStore[fs,fs',paths]
    => RelCalc/dom[fs'.map] = RelCalc/dom[fs.map] - paths and
       FileStoreInvariantVDM[fs']
  }
}
Check_Delete_FileStore: check Delete_FileStore
```

Code 4.50: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_FileStore

The same check is done at the `System` level, just in the same way.

```
assert Delete_System {
  all sys, sys': System, full_path: Path {
    SystemInvariantVDM[sys]                       and
    PathInvariantVDM[full_path]                   and
    pre_FS_DeleteFileDir_System[sys,full_path]    and
    FS_DeleteFileDir_System[sys,sys',full_path]
    => full_path not in RelCalc/dom[sys'.fileStore.map] and
       sys.table = sys'.table                           and
       SystemInvariantVDM[sys']                         and
       RelCalc/dom[sys'.fileStore.map] = RelCalc/dom[sys.fileStore.map] - full_path
  }
}
Check_Delete_System: check Delete_System
```

Code 4.51: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_System

The `FS_DeleteFileDir_Exception` predicate analyses the inputs to generate four possible status values, from which the `FFS_StatusUnknown` should never be returned, provided the conditions necessary for the *_Exception* predicate to be called.

```
assert Delete_Exception_StatusUnknown {
  all sys: System, full_path: Path, status: FFS_StatusUnknown {
```

```
    SystemInvariantVDM[sys]                                  and
    PathInvariantVDM[full_path]                              and
    not (full_path in RelCalc/dom[sys.fileStore.map] and
         pre_FS_DeleteFileDir_System[sys,full_path]      ) and
    FS_DeleteFileDir_Exception[sys,full_path,status]
    => not status = FFS_StatusUnknown
  }
}
Check_Delete_Exception_StatusUnknown: check Delete_Exception_StatusUnknown
```

Code 4.52: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Exception for FFS_StatusUnknown

```
assert Delete_Exception_FileNotFound {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                        and
    PathInvariantVDM[full_path]                    and
    not isElemFileStore[full_path,sys.fileStore]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileNotFound]
  }
}
Check_Delete_Exception_FileNotFound: check Delete_Exception_FileNotFound
```

Code 4.53: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Exception for FS_ErrorFileNotFound

```
assert Delete_Exception_FileStillOpen {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                        and
    PathInvariantVDM[full_path]                    and
    isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileStillOpen]
  }
}
Check_Delete_Exception_FileStillOpen: check Delete_Exception_FileStillOpen
```

Code 4.54: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Exception for FS_ErrorFileStillOpen

```
assert Delete_Exception_DirectoryNonEmpty {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                             and
    PathInvariantVDM[full_path]                         and
    isDirectory[(full_path.(sys.fileStore.map)).info] and
    hasSubFiles[sys.fileStore,full_path]                and
    not isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorDirectoryNonEmpty]
  }
}
Check_Delete_Exception_DirectoryNonEmpty: check Delete_Exception_DirectoryNonEmpty
```

Code 4.55: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Exception for FS_ErrorDirectoryNonEmpty

Finally the `FS_DeleteFileDir_Main` predicate is checked for both success and error cases.

```
assert Delete_Main_Success {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                     and
    PathInvariantVDM[full_path]                 and
    full_path in RelCalc/dom[sys.fileStore.map] and
    pre_FS_DeleteFileDir_System[sys,full_path]  and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys'] and
       status = FFS_StatusSuccess
  }
}
Check_Delete_Main_Success: check Delete_Main_Success
```

Code 4.56: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Main success

```
assert Delete_Main_Error {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                          and
    PathInvariantVDM[full_path]                      and
    not (full_path in RelCalc/dom[sys.fileStore.map] and
         pre_FS_DeleteFileDir_System[sys,full_path]) and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys']       and
       status not = FFS_StatusSuccess
  }
}
Check_Delete_Main_Error: check Delete_Main_Error
```

Code 4.57: Alloy model sliced at FS_DeleteFileDir — Check FS_DeleteFileDir_Main fail

Alloy Analyzer does not find counter examples for any of the above presented assertions.

4.1.6 Model Checking VDM Proof Obligations with the Alloy Analyser

Model Checking POs is done through the translation of the `.pog` file generated by the VDMTools using the `-g` option in command line environment. Translation a VDM PO to Alloy is usually an exercise of translating some initial quantifier expression, enforcing the quantified values invariants, and only then translate the quantified expression.

The VDMTools generated eleven POs for the current specification, from which the first three are subtype POs, and become quite trivial when translated to Alloy.

```
Integrity property #1 :
In function FileSystemLayerBase dirName, file: FileSystemLayerBase.vpp l. 137 c. 46:
    subtype
-------------------------------------------------------------------------------------
(forall full_path : Path &
not (full_path = (<Root>)) =>
 not ((exists [xx_1] : Path &
full_path = [xx_1])) =>
 is_(full_path,seq of FileName))
```

Code 4.58: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 1

```
assert po1 {
  all full_path: Path {
    PathInvariantVDM[full_path]
    => not (full_path = Root
            => not(some xx_1: Path {
                       PathInvariantVDM[xx_1]
                       => full_path = xx_1
                     }))
       => PathInvariantVDM[full_path]
  }
}
Check_PO1: check po1 for 5
```

Code 4.59: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 1

PO2 and PO3 will be skipped, because they are as trivial as the first. PO4 states that the result of the `FS_DeleteFileDir_FileStore` function should preserve the `FileStore` invariant.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 63 c. 9: invariants from FileStore
-------------------------------------------------------------------------------------
(forall fileStore : FileStore, paths : set of Path &
(forall path in set dom (fileStore) &
dirName(path) in set paths =>
 path in set paths) =>
 FileSystemLayerOperations`inv_FileStore(paths <-: fileStore))
```

Code 4.60: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 4

```
assert po4 {
  all fs,fs': FileStore, paths: set Path {
    FileStoreInvariantVDM[fs] and
    PathInvariantVDM[paths]
    => ((((Path - paths)->Path) & iden).(fs.map) in dirName.((Path - paths)->File))
       => fs'.map = fs.map - (paths->paths.(fs.map))
          => FileStoreInvariantVDM[fs']
  }
}
Check_PO4: check po4 for 5
```

Code 4.61: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 4

PO5 is of type Map Application, and is generated at the `FS_DeleteFileDir_Exception` function because of the application of the mapping `sys.fileStore` to the `full_path` parameter. The reason behind this, and other POs of the same type, is the fact that the application of the simple partial relation, the mapping, is only valid to elements of its domain.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_Exception, file:
    FileSystemLayerOperations.vpp l. 87 c. 35: map application
-------------------------------------------------------------------------------------
```

```
(forall sys : System, full_path : Path &
not (not (isElemFileStore(full_path, sys.fileStore))) =>
 not (isElemTablePath(full_path, sys.table)) =>
 full_path in set dom (sys.fileStore))
```

Code 4.62: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 5

```
assert po5 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => not (not (isElemFileStore[full_path, sys.fileStore]))
       => not (isElemTablePath[full_path, sys.table])
          => full_path in RelCalc/dom[sys.fileStore.map]
  }
}
Check_PO5: check po5 for 5
```

Code 4.63: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 5

The next two POs are related to the preservation of the `System` and `FileStore` invariants by the `FS_DeleteFileDir_System` and `FS_DeleteFileDir_FileStore` functions, respectively.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 43 c. 3: invariants from System
-----------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations`inv_System(mu(sys,fileStore|->FS_DeleteFileDir_FileStore(sys.
     fileStore, {full_path}))))
```

Code 4.64: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 6

```
assert po6 {
  all sys,sys': System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FS_DeleteFileDir_FileStore[sys.fileStore,sys'.fileStore,{full_path}] and
          sys'.table = sys.table
          => SystemInvariantVDM[sys']
  }
}
Check_PO6: check po6 for 5 but 2 System
```

Code 4.65: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 6

```
Integrity property #2 :
```

```
In function FileSystemLayerOperations FS_DeleteFileDir_System , file:
    FileSystemLayerOperations.vpp l. 43 c. 51: invariants from FileSystemLayerBase‘
    FileStore
--------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerBase‘inv_FileStore(FS_DeleteFileDir_FileStore(sys.fileStore, {full_path
     })))
```

Code 4.66: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 7

```
assert po7 {
  all sys: System, full_path: Path, fs’: FileStore {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FS_DeleteFileDir_FileStore[sys.fileStore,fs’,{full_path}]
          => FileStoreInvariantVDM[fs’]
  }
}
Check_PO7: check po7 for 5
```

Code 4.67: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 7

Still regarding invariants preservation, PO8 is the preservation of the `FileStore` invariant by the record field selector operator `(.)` used in the `FS_DeleteFileDir_System` function. This PO's expression is quite trivial to discharge, because the invariant is assured before the call to the referred function, and the operator does not produce any changes in the respective value.

```
Integrity property #3 :
In function FileSystemLayerOperations FS_DeleteFileDir_System , file:
    FileSystemLayerOperations.vpp l. 43 c. 55: invariants from FileStore
--------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations‘inv_FileStore(sys.fileStore))
```

Code 4.68: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 8

```
assert po8 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FileStoreInvariantVDM[sys.fileStore]
  }
}
Check_PO8: check po8 for 5
```

Code 4.69: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 8

PO9 checks the correct application of the `FS_DeleteFileDir_FileStore` in its `System` counter part, which is also trivial to discharge because the call to the function is protected by a condition where its pre-condition is required to hold.

```
Integrity property #4 :
In function FileSystemLayerOperations FS_DeleteFileDir_System , file:
    FileSystemLayerOperations.vpp l. 43 c. 51: function application from
    FS_DeleteFileDir_FileStore
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations`pre_FS_DeleteFileDir_FileStore(sys.fileStore , {full_path}))
```

Code 4.70: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 9

```
assert po9 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
  }
}
Check_PO9: check po9 for 5
```

Code 4.71: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 9

PO10 is the same as PO8 although the use of the record fields selector is done in the pre-condition of the `FS_DeleteFileDir_System` function.

```
Integrity property #5 :
In function FileSystemLayerOperations FS_DeleteFileDir_System , file:
    FileSystemLayerOperations.vpp l. 45 c. 39: invariants from FileStore
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) =>
 FileSystemLayerOperations`inv_FileStore(sys.fileStore))
```

Code 4.72: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 10

```
assert po10 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path
```

```
      => FileStoreInvariantVDM[sys.fileStore]
 }
}
Check_PO10: check po10 for 5
```

Code 4.73: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 10

PO11 is similar to PO9, although occurs in the `FS_DeleteFileDir_Main` and the involved invariant belongs to the `System` data type.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 25 c. 35: function application from
    FS_DeleteFileDir_System
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
full_path in set dom (sys.fileStore) and
 pre_FS_DeleteFileDir_System(sys, full_path) =>
 FileSystemLayerOperations`pre_FS_DeleteFileDir_System(sys, full_path))
```

Code 4.74: VDM++ model sliced at FS_DeleteFileDir — Proof Obligation 11

```
assert po11 {
  all sys: System, full_path: Path {
    full_path in RelCalc/dom[sys.fileStore.map] and
    pre_FS_DeleteFileDir_System[sys, full_path]
    => pre_FS_DeleteFileDir_System[sys, full_path]
  }
}
Check_PO11: check po11 for 5
```

Code 4.75: Alloy model sliced at FS_DeleteFileDir — Proof Obligation 11

4.1.7 VDM++ Adapted for the VdmHolTranslator Tool

The VdmHolTranslator supports only a subset of the VDM++ language syntax, so it is necessary to keep the models within the supported syntax. The current VDM++ specification uses several syntactic constructions that need to be rewritten before the translation. The syntactic limitations of the VdmHolTranslator and the respective adaptations made to the original VDM++ model will be described next.

Data type `seq1` is not supported, although is used in the `Path` and `FileName` data types. The data type declarations that use `seq1` can be rewritten using the `seq` data type, and an invariant enforcing the sequence not to be empty.

```
Path = <Root> | seq of FileName
inv path ==
  path <> <Root> => path <> [];
```

Code 4.76: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — Path data type

```
FileName = seq of char
inv fname ==
  fname <> [];
```

Code 4.77: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — FileName data type

Sequence comprehension is not supported and this is the way the `dirName` function is specified. The definition of `dirName` can be rewritten using recursion.

```
dirName : Path -> Path
dirName(full_path) ==
  if   full_path = <Root> or len full_path = 1
  then <Root>
  else let S : seq of FileName = full_path in blast(S);
```

Code 4.78: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — dirName function

This definition of `dirName` could be better written using a `cases` expression, although in that case the translator would have to support patter matching of enumerated type values, which it does not.

```
blast : seq of FileName -> seq of FileName
blast(S) == reverseOrder(tl reverseOrder(S))
pre S <> [];
```

Code 4.79: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — blast function

```
reverseOrder : seq of FileName -> seq of FileName
reverseOrder(S) ==
  if S = []
  then []
  else let h = hd S, t = tl S in
     reverseOrder(t) ^ [h];
```

Code 4.80: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — reverseOrder function

Conditional `elseif` clause is used in the `FS_DeleteFileDir_Exception` function. It is equivalent to have an `if ... then ... else` clause nested in the else branch of another similar conditional expression.

```
FS_DeleteFileDir_Exception: System * Path -> FFS_Status
FS_DeleteFileDir_Exception(sys, full_path) ==
  if   not isElemFileStore(full_path, sys.fileStore)
  then <FS_ErrorFileNotFound>
  else if   isElemTablePath(full_path, sys.table)
       then <FS_ErrorFileStillOpen>
       else if   isDirectory(sys.fileStore(full_path).info) and
                 hasSubFiles(sys.fileStore, full_path)
            then <FS_ErrorDirectoryNonEmpty>
            else <FFS_StatusUnknown>;
```

Code 4.81: VDM++ model adapted to HOL translation, sliced at FS_DeleteFileDir — FS_DeleteFileDir_Exception function

Function `is_` is used in three POs, to check for subtype constraints. A semantically equivalent construction could be done using the type's invariant predicate, whenever it exists. Whenever the `is_` function is used for a type that has no invariant, the PO will be discarded.

4.1.8 Correcting the Translated HOL4 Model

The output of the VdmHolTranslator is a HOL model with some errors. Some of those errors are explained in the Open Issues section of the tool's web site [91], which also provides solutions. However, there are still errors for which solutions will be presented here.

Data type `Path`**.** HOL interpreter throws an exception because it does not recognize the type `((char list) list)`. A type constructor can be used to make it accept the type. This alteration will have to be enforced throughout the model whenever a `Path` us used.

```
Hol_datatype ‘Path =  RootQuoteLiteral | ((char list) list) ‘;
Define ‘inv_Path (inv_Path_subj: RootQuoteLiteral | ((char list) list) )  =
  (let path = inv_Path_subj in
   (((\x y . ~ (x = y)) path RootQuoteLiteral)
   ==> ((\x y . ~ (x = y)) path []) ))‘;
```

Code 4.82: Translated HOL model — data type Path

```
Hol_datatype ‘Path =  RootQuoteLiteral | AList of ((char list) list) ‘;
Define ‘inv_Path (inv_Path_subj: Path) =
  (let path = inv_Path_subj in
   (\x y. ~(x = y)) path RootQuoteLiteral
   ==> (\x y. ~(x = y)) path (AList []))‘;
```

Code 4.83: Corrected HOL model — data type Path

Function `reverseOrder`**.** HOL interpreter throws an exception because it can not prove totality of the recursive function `reverseOrder`. Redefining the function in two separate branches, termination and recursive, is a possible workaround.

```
Define ‘reverseOrder (reverseOrder_parameter_1:((char list) list))  =
  (let S = reverseOrder_parameter_1 in
   (if   (S  = [] )
    then []
    else (let h = (HD S) and t = (TL S) in ((reverseOrder t)  ++ [h] ))))‘;
```

Code 4.84: Translated HOL model — function reverseOrder

```
Define ‘(reverseOrder [] = []) /\
        !h t. reverseOrder (h::t) = reverseOrder t ++ [h]‘;
```

Code 4.85: Corrected HOL model — function reverseOrder

Function `dirName`**.** There are multiple problems in the translated definition of `dirName`. The interpreter detects right away the type error in the application of `LENGTH` to a value of type `Path`, and there is also the issue of the new type constructor included in the `Path` data type definition.

```
Define ‘dirName (dirName_parameter_1:Path)  =
  (let full_path = dirName_parameter_1 in
   (if ((full_path  = RootQuoteLiteral )  \/ ((LENGTH full_path)  = 1 ) )
    then RootQuoteLiteral
    else (let S = full_path in (blast S))))‘;
```

Code 4.86: Translated HOL model — function reverseOrder

```
Define ‘dirName (path:Path) =
  case path of
     RootQuoteLiteral -> RootQuoteLiteral
  || AList fileNames  -> (if   (TL fileNames) = []
                          then RootQuoteLiteral
                          else AList (blast fileNames))‘;
```

Code 4.87: Corrected HOL model — function reverseOrder

A sequence having size one is semantically equivalent to the tail of the same sequence being the empty sequence, although it seems better for the proof not to use the `LENGTH` operator. This is based on the fact that during some experiences with HOL it did not assume the equivalence between the two predicates.

Function `inv_FileStore`**.** This is one of the errors that is covered in the Open Issues of the translator's web site. It is related with the translation of mappings application, which in HOL requires the operator `FAPPLY`, or its infix equivalent (**'**).

```
Define ‘inv_FileStore (inv_FileStore_subj:((Path, File) fmap))  =
  (let fileStore = inv_FileStore_subj in
   (!  uni_1_var_1.((((uni_1_var_1  IN (FDOM fileStore) )  /\ (?  path.(path  =
       uni_1_var_1 )) )  /\ T )     ==> (let path = uni_1_var_1 in
       (let parent = (dirName path) in
       ((parent  IN (FDOM fileStore) )  /\ (isDirectory (fileStore parent).info) ))
          ) )))‘;
```

Code 4.88: Translated HOL model — function inv_FileStore

```
Define ‘inv_FileStore (inv_FileStore_subj:((Path, File) fmap)) =
  (let fileStore = inv_FileStore_subj in
   !uni_1_var_1.
   (uni_1_var_1 IN FDOM fileStore /\ ?path. path = uni_1_var_1) /\ T
    ==> (let path = uni_1_var_1 in
         let parent = dirName path in
         parent IN FDOM fileStore /\ isDirectory (fileStore ’ parent).info))‘;
```

Code 4.89: Corrected HOL model — function inv_FileStore

Function `FS_DeleteFileDir_Exception`**.** Just such as in the previous situation, a correct definition can be obtained using the `FAPPLY` operator.

```
Define ‘FS_DeleteFileDir_Exception (FS_DeleteFileDir_Exception_parameter_1:System)
                                   (FS_DeleteFileDir_Exception_parameter_2:Path)  =
  (let sys = FS_DeleteFileDir_Exception_parameter_1 and
```

```
      full_path = FS_DeleteFileDir_Exception_parameter_2 in
  (if (~ (isElemFileStore full_path sys.fileStore))
   then FS_ErrorFileNotFoundQuoteLiteral
   else (if (isElemTablePath full_path sys.table)
         then FS_ErrorFileStillOpenQuoteLiteral
         else (if ((isDirectory (sys.fileStore full_path).info)  /\
                   (hasSubFiles sys.fileStore full_path) )
         then FS_ErrorDirectoryNonEmptyQuoteLiteral
         else FFS_StatusUnknownQuoteLiteral))))`;
```

Code 4.90: Translated HOL model — function FS_DeleteFileDir_Exception

```
Define `FS_DeleteFileDir_Exception (FS_DeleteFileDir_Exception_parameter_1:System)
                                   (FS_DeleteFileDir_Exception_parameter_2:Path)
                                        =
  (let sys = FS_DeleteFileDir_Exception_parameter_1 and full_path =
      FS_DeleteFileDir_Exception_parameter_2 in
   (if   ~(isElemFileStore full_path sys.fileStore)
    then FS_ErrorFileNotFoundQuoteLiteral
    else (if   isElemTablePath full_path sys.table
          then FS_ErrorFileStillOpenQuoteLiteral
          else (if   (isDirectory (sys.fileStore ' full_path).info) /\
                     (hasSubFiles sys.fileStore full_path)
                then FS_ErrorDirectoryNonEmptyQuoteLiteral
                else FFS_StatusUnknownQuoteLiteral))))`;
```

Code 4.91: Corrected HOL model — function FS_DeleteFileDir_Exception

4.1.9 Discharging VDM Proof Obligations with HOL4

With a correct HOL model is possible to execute the proof commands generated by the VdmHolTranslator. Due to the adaptations made on the VDM++ model, the POs that got translated to HOL proof commands, are not exactly the same as the POs presented earlier, in Section 4.1.4. The adapted VDM++ model yields even more POs, mainly because of the redefinition of `dirName`, introducing two new functions. The equivalence between the initial POs and the POs that can now be discharged in HOL, is assured by the equivalence of the two VDM++ models. However, no proof of equivalence between models have been made, the assumption relies on the knowledge of the semantics of both VDM++ and Alloy, and the conviction that they are in fact equivalent. There are some other POs of the initial VDM++ model that have no meaning in the adapted model. This is the case of PO1, PO2 and PO3, that are for subtype checking related with the use of `seq1` data type, that is no longer used in the adapted model.

Proof Obligation 4: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 5: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 6: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 7: there is no match for PO7 in the adapted model, although PO7 is equivalent to PO4, since in the latter the body of the `FS_DeleteFileDir_FileStore` function is used, and in the first the body is replaced by the function call.

Proof Obligation 8: is almost equal to PO7, differing only in the class that is analyzed by the integrity checked. This is the result of specifying the base data types in the `FileSystemLayerBase` class, and the behavior in the `FileSystemLayerOperations` class.

Proof Obligation 9: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 10: is not generated in the adapted model, although is quite trivial to discharge given that it consists of an implication where the consequent is included in the hypothesis.

Proof Obligation 11: mapped to a PO in the adapted model, and discharged with HOL.

All original POs are considered discharged, and the corresponding VDM++ model verified to be correct.

4.2 FS_OpenFileDir

This operation creates a file or directory, or opens existing files. It is specified in page 39 of the IFFSCRG document. Figure 4.2 reveals that it is much more complex than the previous operation, note that it requires an entire page just for the signature and parameters description.

4.2.1 Requirements Analysis

Some of the operation's parameters will be part of the specifications and other will be abstracted either because they deal with features that are not part of the specification, or because their values can be calculated from the specification. The parameters `share_mode`, `ram_buffer_mode`, and `trans_flag` will be abstracted because concurrency, RAM buffers and transactions, respectively, are not part of the specification. As for the `static_info_type` it will also be abstracted because the file types can be obtained by inspection of the respective files.

The operation will create and/or open files, or create a directory depending on the open mode. Its result will be the status and possibly a handler to reference a file.

R1 THE OPERATION CREATES FILES AND DIRECTORIES.

R2 THE OPERATION OPENS FILES.

R3 WHENEVER A FILE IS OPENED THE OPERATION SHALL RETURN AN HANDLE TO THAT FILE.

R4 THE RETURNED HANDLERS SHALL REFERENCE AN OPEN FILE.

The open mode will specify the kind of action to be taken, and the kind of access that will be granted to the returned file handle.

R5 THERE SHOULD BE A WAY TO SPECIFY THE OPEN MODE.

R6 THERE SHOULD BE A WAY TO RESTRICT THE ACCESS ON OPEN FILES.

File System API Reference

4.14 FS_OpenFileDir

Creates a directory, or creates and opens a file, or opens existing file

Syntax

```
FFS_Status      FS_OpenFileDir (
        mOS_char              *full_path,
        FS_FileHandlePtr      file_handle_ptr,
        UINT32                attributes,
        FS_OpenMode           open_mode,
        FS_ShareMode          share_mode,
        FS_RamBufferMode      ram_buffer_mode,
        UINT8                 trans_flag,
        UINT8                 static_info_type );
```

Parameters

Parameter	Description
*full_path	(IN) The fullpath of the filename for the file or directory. This function fails with an error if this input is NULL.
file_handle_ptr	(OUT) Pointer to an open file handle (FS_FileHandle) that is owned by the calling function. Upon exit from this function, the variable pointed to by this input is assigned a unique file handle. This handle is used in subsequent calls to File System functions to reference the file that was opened or created. This function returns with an error if this input is NULL. (See Section 15.2.7, "FS_OpenFileInfo" on page 258.)
attributes	(IN) File or directory attributes to be set. No effort is made to interpret these.
open_mode	(IN) Specifies whether the file is to be opened or created, and what kind of access will be granted by the resulting file handle. This input must be one of the following FS_OpenMode constants (See Section 15.2.8, "FS_FindMode #Defines" on page 261): • FS_CreateNew: Attempts to create a new file with the given name in the specified path. Returns with an error if the file already exists. If successful, the file is opened for read/write access. • FS_CreateAlways: Creates the file with the given name in the specified path. If a file with the same name already exists, it is deleted and replaced by the new file. If successful, the file is opened for read/write access. • FS_OpenRead: Attempts to open an existing file for read-only access. Returns with an error if the file does not exist. Any attempts to write to a file opened in this mode will fail with an error. • FS_OpenWrite: Attempts to open an existing file for read/write access. Returns with an error if the file does not exist. • FS_OpenAlways: Attempts to open an existing file for read/write access. If the file does not exist, it is created.
share_mode	(IN) File sharing mode, FS_ShareMode. This parameter is ignored when the static_info_type is for a directory. The share_mode parameter of type FS_ShareMode defines all allowable modes in which a file can be shared when using the FS_OpenFileDir function. See Section 15.2.15, "FS_ShareMode #Defines" on page 263.

Intel® Flash File System Core Reference Guide 39

Figure 4.2: `FS_OpenFileDir` operation (Permission to reproduce this excerpt is kindly granted by Intel Corporation).

R7 THERE SHOULD BE A WAY TO MAP OPEN MODES TO ACCESS MODES.

Detail information about each open mode is given at the `omode` parameter description, and will be denoted in the next requirements:

`FS_CreateNew` has the requirements:

R8 ATTEMPTS TO CREATE A FILE POINTED BY `FULL_PATH`.

R9 RETURNS AN ERROR IF THE FILE ALREADY EXISTS.

R10 WHEN SUCCESSFUL THE FILE IS OPENED WITH READ AND WRITE ACCESS GRANTED.

`FS_CreateAlways` has the requirements:

R11 ATTEMPTS TO CREATE A FILE POINTED BY `FULL_PATH`, IF A FILE ALREADY EXISTS IT IS DELETED FIRST.

R12 IF SUCCESSFUL THE FILE IS OPENED WITH READ AND WRITE ACCESS GRANTED.

`FS_OpenRead` has the requirements:

R13 ATTEMPTS TO OPEN AN EXISTING FILE POINTED BY `FULL_PATH`.

R14 RETURNS AN ERROR IF THE FILE DOES NOT EXIST.

R15 IF SUCCESSFUL THE FILE IS OPENED WITH READ ONLY ACCESS GRANTED.

`FS_OpenWrite` has the requirements:

R16 ATTEMPTS TO OPEN AN EXISTING FILE POINTED BY `FULL_PATH`.

R17 RETURNS AN ERROR IF THE FILE DOES NOT EXIST.

R18 IF SUCCESSFUL THE FILE IS OPENED WITH READ AND WRITE ACCESS GRANTED.

`FS_OpenAlways` has the requirements:

R19 ATTEMPTS TO CREATE A FILE POINTED BY `FULL_PATH`, IF A FILE ALREADY EXISTS IT IS DELETED FIRST.

R20 IF SUCCESSFUL THE FILE IS OPENED WITH READ AND WRITE ACCESS GRANTED.

There is another possible value for the open mode parameter that is not covered in the parameter description, which is `FS_CreateAlwaysReadOnly`. The behavior for this possible value is extrapolated from its designation.

AR1 ATTEMPTS TO CREATE A FILE POINTED BY `FULL_PATH`, IF A FILE ALREADY EXISTS IT IS DELETED FIRST.

AR2 IF SUCCESSFUL THE FILE IS OPENED WITH READ ACCESS GRANTED.

From the omode parameter analysis there is one other requirement regarding access violations.

R21 IF THE GRANTED ACCESS IS READ ONLY, SUBSEQUENT ATTEMPTS TO WRITE TO THE FILE WILL FAIL WITH AN ERROR.

When opening a file a new structure FS_OpenFileInfo is created, and the fileOffset fields as to be set according to the kind of access that is being made on the file. It is said in the operation description that a new file should have an offset of zero.

R22 FOR READ ACCESS THE FILE OFFSET SHALL BE SET TO THE BEGINNING OF THE FILE.

R23 FOR WRITE ACCESS THE FILE OFFSET SHALL BE SET TO THE END OF THE FILE.

R24 FOR NEWLY CREATED FILES THE OFFSET SHALL BE ZERO.

There are more additional requirements regarding the open modes. As mentioned before, the operation shall open files although not directories, there are open modes specific for both creating and opening, so what should happen when the operation is called on a directory with a opening mode specified? To answer the question it is necessary to impose more requirements regarding the issue.

AR3 CALLS ON DIRECTORIES WITH OPEN SPECIFIC MODES SHALL RESULT IN AN ERROR.

4.2.2 VDM++ Model

The requirements analysis found twenty seven requirements for the FS_OpenFileDir operation, form which six are only related with data structures: **R5**, **R6**, **R21**, **R22**, **R23**, and **R24**; nineteen are only related with the behavior of the operation: **R1**, **R2**, **R3**, **R8**, **R9**, **R10**, **R11**, **R12**, **R13**, **R14**, **R15**, **R16**, **R17**, **R18**, **R19**, **R20**, **AR1**, **AR2**, and **AR3**; two**R4** and **R7** that are related to both data structures and behavior of the operation.

Starting with the data structures requirements, **R5** and **R6** are covered with the introduction of two new structures specified in the IFFSCRG document, namely FS_OpenMode (page 261 section 15.2.9 FS_OpenMode #Defines) and FS_AccessMode (page 262 Section 15.2.10 FS_AccessMode #Defines).

```
FS_OpenMode =
    <FS_CreateNew>
  | <FS_CreateAlways>
  | <FS_OpenRead>
  | <FS_OpenWrite>
  | <FS_OpenAlways>
  | <FS_OpenWriteOnly>
  | <FS_CreateAlwaysReadOnly>
  | <FS_CreateNewReadOnly>;
```

Code 4.92: VDM++ model sliced at FS_OpenFileDir — FS_OpenMode data type

```
FS_AccessMode =
    <FS_AccessReadOnly>
  | <FS_AccessWriteOnly>
  | <FS_AccessReadWrite>;
```

Code 4.93: VDM++ model sliced at FS_OpenFileDir — FS_AccessMode data type

Requirement **R7** states the need to map the supplied open modes to the respective access mode. This could be accomplished through a function, although the mapping data type suits the purpose better.

```
fs_open2access_mode_map: map FS_OpenMode to FS_AccessMode =
{
  <FS_CreateNew>             |-> <FS_AccessReadWrite>,
  <FS_CreateAlways>          |-> <FS_AccessReadWrite>,
  <FS_OpenRead>              |-> <FS_AccessReadOnly>,
  <FS_OpenWrite>             |-> <FS_AccessReadWrite>,
  <FS_OpenAlways>            |-> <FS_AccessReadWrite>,
  <FS_OpenWriteOnly>         |-> <FS_AccessWriteOnly>,
  <FS_CreateAlwaysReadOnly>  |-> <FS_AccessReadOnly>,
  <FS_CreateNewReadOnly>     |-> <FS_AccessReadOnly>
};
```

Code 4.94: VDM++ model sliced at FS_OpenFileDir — fs_open2access_mode_map mapping

The mapping is constructed by inspecting the requirements regarding the open mode parameter, and extrapolating the values which are not specified. As for requirements **R22**, **R23**, and **R24** they are met by introducing a new data type `FileContents`,

```
FileContents = seq of token;
```

Code 4.95: VDM++ model sliced at FS_OpenFileDir — FileContents data type

and new fields in existing record data types. It is necessary to introduce data `contents` in files to calculate the appropriate offset when opening a file. The fact that files now have `contents` introduces the need for an invariant stating that regular files have contents, and directories do not. This does not follow form POSIX nor from the IFFSCRG documentation, it is a conceptual constraint derived from the mapping between paths and files. If the file store was represented through a tree shape structure, then contents of directories would be their sub-files' names.

```
File ::
  info     : FS_FileDirInfo
  contents : [FileContents]
inv file ==
  (isDirectory(file.info)   and file.contents =  nil) or
  (isRegularFile(file.info) and file.contents <> nil);
```

Code 4.96: VDM++ model sliced at FS_OpenFileDir — File data type

There is also the need to store more information on open files, and to do that the fields `fileOffset` and `accessMode` are introduced in the `FS_OpenFileInfo` record. It is considered that keeping track

of the file offset is needed just for open files, hence not including this field in the FS_FileDirInfo record. However this can be done if it becomes necessary.

```
FS_OpenFileInfo ::
  fileOffset : nat1
  accessMode : FS_AccessMode
  path       : Path;
```

Code 4.97: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileInfo data type

The one requirement that will not be completely expressed in the specification at this point is **R21**, and this is because it needs to be enforced in the write related operations. Even though the requirement can be accomplished through data structures, things can only be prepared for future operations to have a way to query the system about granted access to files. This query is done by inspecting the respective FS_OpenFileInfo accessMode field.
As for requirement **R4** it is already covered by the previous System definition, which has an invariant assuring referential integrity over fileStore and table. However to assure requirement **R2**, that states that only files can be opened, and extra clause is added.

```
System :: table     : OpenFilesTable
          fileStore : FileStore
inv sys ==
  forall ofi in set rng sys.table &
    isElemFileStore(ofi.path, sys.fileStore) and
    isRegularFile(sys.fileStore(ofi.path).info);
```

Code 4.98: VDM++ model sliced at FS_OpenFileDir — System data type

The operation will span over the main three data types System, FileStore and OpenFilesTable, so it will have three functions regarding the respective data types manipulation, a top level *_Main* total function and the error handling *_Exception* function. The main function will have to check if the necessary conditions to call the *_System* function hold, and the possible error conditions regarding the open mode parameter. This covers requirements **R9**, **R14**, **R17** and **AR3**.

```
checkOpenMode: System * Path * FS_OpenMode -> bool
checkOpenMode(sys, path, omode) ==
  not (isCreateNew(omode)    and isElemFileStore(path, sys.fileStore))     and
  not (isCreateAlways(omode) and isElemTablePath(path, sys.table))         and
  not (isOpen(omode)         and not isElemFileStore(path, sys.fileStore)) and
  not ((isOpen(omode) or omode = <FS_OpenAlways>) and
       isElemFileStore(path, sys.fileStore)       and
       not isRegularFile(sys.fileStore(path).info));
```

Code 4.99: VDM++ model sliced at FS_OpenFileDir — checkOpenMode function

If the pre-condition of the FS_DeletefileDir_System and the conditions on open mode parameter hold, then the *_Main* function will have to check if it is necessary to remove the file prior to its creation according requirements **R11** and **AR1**, and consequently call the FS_DeleteFileDir operation to remove the specified file.

```
mustDeleteFirst: FileStore * Path * FS_OpenMode -> bool
mustDeleteFirst(fs, path, omode) ==
  isCreateAlways(omode) and isElemFileStore(path, fs);
```

Code 4.100: VDM++ model sliced at FS_OpenFileDir — mustDeleteFirst function

There is a detail in the delete condition, that is related with the root directory. The problem is that whenever the operation is invoked with the set of parameters that will lead to the deletion of the root directory and its consequent creation, it is necessary to enforce that the attributes used to create the root directory specify it to be a directory. This is the result of model checking in Alloy, which provided a counter example where the `FileStore` invariant was violated by the open operation. Even though the pre-condition of the `FS_OpenFileDir_FileStore` already enforces the necessary requirements to create the root directory, it does not protect against the case when it is deleted and recreated, because in this scenario when the pre-condition is tested the root directory exists in the `fileStore`, hence the pre-condition will hold.

```
FS_OpenFileDir_Main: System * Path * Attributes * FS_OpenMode
  -> System * [FS_FileHandle] * FFS_Status
FS_OpenFileDir_Main(sys, full_path, attributes, omode) ==
  if   pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
       checkOpenMode(sys, full_path, omode)
  then if   mustDeleteFirst(sys.fileStore, full_path, omode) and
            (full_path = <Root> => attributes.fileType = <Directory>)
       then let mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path) in
            if   status <> <FFS_StatusSuccess>
            then mk_(sys', nil, status)
            else let result = FS_OpenFileDir_System(sys', full_path, attributes, omode)
                in
                       mk_(result.#1, result.#2, <FFS_StatusSuccess>)
       else let mk_(sys'',handle) = FS_OpenFileDir_System(sys, full_path, attributes,
           omode) in
            mk_(sys'', handle, <FFS_StatusSuccess>)
  else mk_(sys, nil, FS_OpenFileDir_Exception(sys, full_path, omode));
```

Code 4.101: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir_Main function

The flow inside the *_Main* function continues to the call of the next function in the cascade, that might be the *_Exception* function if an error is to be returned, or the *_System* function if the operation is to be carried out.

```
FS_OpenFileDir_System: System * Path * Attributes * FS_OpenMode
  -> System * [FS_FileHandle]
FS_OpenFileDir_System(sys, full_path, attr, omode) ==
  let fileStore'        = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset            = getOpenOffset(fileStore'(full_path), omode),
      table'            = table ++ if   handle <> nil and isElemTableHandle(handle,
          table)
                                   then {handle |-> mu(table(handle), fileOffset |->
                                       offset)}
                                   else {|->} in
  mk_(mk_System(table', fileStore'), handle)
pre pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr);
```

Code 4.102: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir_System function

The `FS_OpenFileDir_System` function has a pre-condition enforcing the necessary conditions to call the subsequent functions *_FileStore* and *_Table*, that will modify the `fileStore` and `table` attributes respectively. However, due to requirements **R22** and **R23** it is necessary to update the `FS_OpenFileInfo` value that the *_Table* function might create. This is done at the `System` level so that file storage and open files keep logically separated. Still regarding file offset, the document states that new files should have an offset set to zero, and this is interpreted as having the file offset pointing to the element that will be written next, because it is an empty file. The zero value is often the first index of an array, although in VDM (both SL and ++) the first index of a sequence is one, so the requirement **R24** is accomplished using the value one instead of zero.

```
getOpenOffset: File * FS_OpenMode -> [nat1]
getOpenOffset(file, omode) ==
  if   file.contents <> nil
  then cases omode:
         <FS_OpenWrite>  -> (len file.contents) + 1,
         <FS_OpenAlways> -> (len file.contents) + 1,
         others          -> 1
       end
  else nil;
```

Code 4.103: VDM++ model sliced at FS_OpenFileDir — getOpenOffset function

The *_FileStore* function will just have to create new files, covering requirements **R1**, **R8**, **R11**, **R13**, **R16**, **R19**, and **AR1**. The pre-condition allows two types of invocations: create the root directory with the correct attributes, or create any kind of file and directory which has a valid parent directory.

```
FS_OpenFileDir_FileStore: FileStore * Path * Attributes -> FileStore
FS_OpenFileDir_FileStore(fileStore, full_path, attributes) ==
  if   not isElemFileStore(full_path, fileStore)
  then let content = emptyFileContents(attributes.fileType),
           newFile = mk_File(newFileDirInfo(attributes), content) in
       fileStore munion { full_path |-> newFile }
  else fileStore
pre (full_path = <Root> and attributes.fileType = <Directory>) or
    (let parent = dirName(full_path) in
     isElemFileStore(parent,fileStore) and isDirectory(fileStore(parent).info));
```

Code 4.104: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir_FileStore function

Opening files is something that occurs at the `OpenFilesTable` level, and the respective function covers the requirements **R2**, **R3** and **R4**.

```
FS_OpenFileDir_Table: OpenFilesTable * Path * FS_OpenMode * FileType -> OpenFilesTable *
     [FS_FileHandle]
FS_OpenFileDir_Table(table, full_path, omode, fileType) ==
  if   fileType = <Directory>
  then mk_(table, nil)
  else let amode  = fs_open2access_mode_map(omode),
           ofi    = mk_FS_OpenFileInfo(1, amode, full_path),
           handle = newFileHandle(dom table) in
       mk_(table munion { handle |-> ofi }, handle);
```

Code 4.105: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir_Table function

The error handling function takes in account the remaining requirements that are captured in in the `FS_OpenFileDir_Excepetion`.

```
FS_OpenFileDir_Exception: System * Path * FS_OpenMode -> FFS_Status
FS_OpenFileDir_Exception(sys, full_path, omode) ==
  if      isCreateNew(omode) and isElemFileStore(full_path, sys.fileStore)
  then    <FS_ErrorFileAlreadyExists>
  elseif isCreateAlways(omode) and isElemTablePath(full_path, sys.table)
  then    <FS_ErrorFileStillOpen>
  elseif isOpen(omode) and not isElemFileStore(full_path, sys.fileStore)
  then    <FS_ErrorFileNotFound>
  elseif isOpen(omode) and isElemFileStore(full_path, sys.fileStore) and
         not isRegularFile(sys.fileStore(full_path).info)
  then    <FFS_StatusInvalidParameter>
  elseif full_path <> <Root> and not isElemFileStore(dirName(full_path), sys.fileStore)
  then    <FS_ErrorInvalidPath>
  elseif full_path <> <Root> and not isDirectory(sys.fileStore(dirName(full_path)).info)
  then    <FS_ErrorInvalidPath>
  else    <FFS_StatusUnknown>;
```

Code 4.106: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir_Exception function

```
FFS_Status =
    <FFS_StatusSuccess>
  | <FS_ErrorFileNotFound>
  | <FS_ErrorFileStillOpen>
  | <FS_ErrorDirectoryNonEmpty>
  | <FS_ErrorFileAlreadyExists>
  | <FS_ErrorInvalidPath>
  | <FFS_StatusInvalidParameter>
  | <FFS_StatusUnknown>;
```

Code 4.107: VDM++ model sliced at FS_OpenFileDir — FFS_Status data type

The objectified operation is specified as `FS_OpenFileDir`.

```
FS_OpenFileDir : Path * Attributes * FS_OpenMode
  ==> [FS_FileHandle] * FFS_Status
FS_OpenFileDir(full_path, attributes, omode) ==
  def mk_(sys',handle,status)
        = FS_OpenFileDir_Main(sys, full_path, attributes, omode) in
  (sys := sys'; return mk_(handle, status));
```

Code 4.108: VDM++ model sliced at FS_OpenFileDir — FS_OpenFileDir operation

4.2.3 Unit Testing the VDM++ Model

Just such as it was done for the `FS_DeleteFileDir` operation, testing will be done at the different data types levels and at the outmost level. In the specification of `FS_OpenFileDir` the `OpenFilesTable` data type also takes part of the operation, and shall be included in the test cases.

At the `FileStore` data type level one of two things can happen (1) the file pointed by path does not exist and must be created, or (2) the file does exist and the file store must remain as it was. This makes it useful to specify two different test methods `CreateOpenTest` for (1) and `DoNothingOpenTest` for (2).

```
CreateOpenTest: Path * Attributes ==> ()
CreateOpenTest(path,attr) ==
  let fs' = FS_OpenFileDir_FileStore(fs,path,attr) in
  (AssertTrue(path in set dom fs');
   AssertTrue(fs'(path).info.attributes = attr);
   fs := fs');
```

Code 4.109: Tests for VDM++ model sliced at FS_OpenFileDir — CreateOpenTest at FileStore level

For the case where files are created assertions are made to check if the respective file was in fact created, with the supplied attributes.

```
DoNothingOpenTest: Path * Attributes ==> ()
DoNothingOpenTest(path,attr) ==
  let fs' = FS_OpenFileDir_FileStore(fs,path,attr) in
  (AssertTrue(fs = fs');
   fs := fs');
```

Code 4.110: Tests for VDM++ model sliced at FS_OpenFileDir — DoNothingOpenTest at FileStore level

For the case where no action is performed the assertions must ensure exactly that. The overall test case is as follows.

```
RunTest: () ==> ()
RunTest() ==
  (CreateOpenTest(<Root>, vals.dirAttr);
   CreateOpenTest(["etc"], vals.dirAttr);
   CreateOpenTest(["bin"], vals.dirAttr);
   CreateOpenTest(["etc", "hosts"], vals.fileAttr);
   CreateOpenTest(["etc", "conf.d"], vals.fileAttr);
   CreateOpenTest(["etc", "resolv.conf"], vals.fileAttr);
   CreateOpenTest(["bin", "ls"], vals.fileAttr);
   CreateOpenTest(["bin", "wc"], vals.fileAttr);

   DoNothingOpenTest(<Root>, vals.dirAttr);
   DoNothingOpenTest(["etc"], vals.dirAttr);
   DoNothingOpenTest(["bin"], vals.dirAttr);
   DoNothingOpenTest(["etc", "hosts"], vals.fileAttr);
   DoNothingOpenTest(["etc", "conf.d"], vals.fileAttr);
   DoNothingOpenTest(["etc", "resolv.conf"], vals.fileAttr);
   DoNothingOpenTest(["bin", "ls"], vals.fileAttr);
   DoNothingOpenTest(["bin", "wc"], vals.fileAttr));
```

Code 4.111: Tests for VDM++ model sliced at FS_OpenFileDir — Test case at FileStore level

Testing the `FS_OpenFileDir_Table` function is done in one test method that asserts the different behavior of the function when dealing with files or directories. In the case of files the returned handler must not be `nil`, there must be an `FS_OpenFileInfo` value which has the specified `full_path` in the `path` field; the resulting table's domain must be equal to the original table's domain unified with the singleton set containing the new handler; the cardinality of the resulting table should be equal to the cardinality of the original table plus one, hence assuring a new handler was created; the path to file referenced by the returned handler must be the same specified in the `full_path` parameter of the function; and the access mode for the opened file should be according to the mapping between open modes and access modes.

Name	#Calls	Coverage
FileSystemLayerOperations‘FS-DeleteFileDir-Main	2	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-System	1	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-FileStore	1	$\surd$
FileSystemLayerOperations‘FS-DeleteFileDir-Exception	1	89%
FileSystemLayerOperations‘FS-OpenFileDir-Main	11	$\surd$
FileSystemLayerOperations‘FS-OpenFileDir-System	12	$\surd$
FileSystemLayerOperations‘FS-OpenFileDir-FileStore	28	$\surd$
FileSystemLayerOperations‘FS-OpenFileDir-Table	18	$\surd$
FileSystemLayerOperations‘FS-OpenFileDir-Exception	7	98%
FileSystemLayerOperations‘getOpenOffset	20	$\surd$
FileSystemLayerOperations‘mustDeleteFirst	4	$\surd$
FileSystemLayerOperations‘checkOpenMode	9	$\surd$
Total Coverage		**99%**

Table 4.8: Test coverage for functional specification of `FS_OpenFileDir`

Name	#Calls	Coverage
FileSystemLayerObj‘FS-OpenFileDir	8	$\surd$
Total Coverage		**58%**

Table 4.10: Test coverage for objectified specification of `FS_OpenFileDir`

```
SimpleOpenTest: Path * FS_OpenMode * FileType ==> ()
SimpleOpenTest(path,omode, fileType) ==
  let mk_(table',handle) = FS_OpenFileDir_Table(table, path, omode, fileType) in
  (if fileType = <RegularFile>
   then (AssertTrue(handle <> nil);
         AssertTrue(path in set { ofi.path | ofi in set rng table' });
         AssertTrue(dom table' = dom table union {handle});
         AssertTrue(card dom table' = card dom table + 1);
         AssertTrue(path = table'(handle).path);
         AssertTrue(table'(handle).accessMode = fs_open2access_mode_map(omode));
        )
   else AssertTrue(handle = nil);
```

Code 4.112: Tests for VDM++ model sliced at FS_OpenFileDir — SimpleOpenTest at OpenFilesTable level

For directories the test method simply asserts that the resulting handler is `nil`. The test case is as follows.

```
RunTest: () ==> ()
RunTest() ==
  (dcl init         : OpenFilesTable := table;

   SimpleOpenTest(["bin"], <FS_CreateNew>, <Directory>);
   SimpleOpenTest(["etc","hosts"], <FS_CreateNew>, <RegularFile>);
   SimpleOpenTest(["etc","conf.d"], <FS_CreateNew>, <Directory>);
   SimpleOpenTest(<Root>, <FS_OpenRead>, <Directory>);
```

```
  SimpleOpenTest(["etc"], <FS_OpenRead>, <Directory>);
  SimpleOpenTest(["bin","ls"], <FS_CreateNew>, <RegularFile>);

  AssertTrue(card dom table = card dom init + 2));
```

Code 4.113: Tests for VDM++ model sliced at FS_OpenFileDir — Test case at Table level

At the `System` level the test method checks for the same properties as its equivalent in `FileStore` and `OpenFilesTable`, in addition it checks if the `fileOffest` filed is assigned correctly by the `Fs_OpenFileDir_System` function.

```
SimpleOpenTest: Path * Attributes * FS_OpenMode ==> ()
SimpleOpenTest(path, attr, omode) ==
  let mk_(sys', handle) = FS_OpenFileDir_System(sys, path, attr, omode) in
  (AssertTrue(path in set dom sys'.fileStore);
   AssertTrue(sys'.fileStore(path).info.attributes = attr);
   if attr.fileType = <RegularFile>
   then (AssertTrue(handle <> nil);
         AssertTrue(path in set { ofi.path | ofi in set rng sys'.table });
         AssertTrue(dom sys'.table = dom sys.table union {handle});
         AssertTrue(card dom sys'.table = card dom sys.table + 1);
         AssertTrue(path = sys'.table(handle).path);
         AssertTrue(sys'.table(handle).accessMode = fs_open2access_mode_map(omode));
         AssertTrue(path in set dom sys'.fileStore);
         AssertTrue(sys'.fileStore(path).info.attributes = attr);
         AssertTrue(sys'.table(handle).fileOffset = getOpenOffset(sys'.fileStore(path),
             omode))
        )
   else AssertTrue(handle = nil);
   h := handle;
   sys := sys');
```

Code 4.114: Tests for VDM++ model sliced at FS_OpenFileDir — SimpleOpenTest at System level

The overall test case is as follows.

```
RunTest: () ==> ()
RunTest() ==
  (SimpleOpenTest(<Root>, vals.dirAttr, <FS_CreateAlways>);
   SimpleOpenTest(<Root>, vals.dirAttr, <FS_OpenAlways>);
   SimpleOpenTest(["etc"], vals.dirAttr, <FS_CreateNew>);
   SimpleOpenTest(["bin"], vals.dirAttr, <FS_CreateAlways>);
   SimpleOpenTest(["etc","hosts"], vals.fileAttr, <FS_OpenWrite>);
   SimpleOpenTest(["etc","conf.d"], vals.fileAttr, <FS_OpenRead>);
   SimpleOpenTest(["bin","ls"], vals.fileAttr, <FS_OpenWriteOnly>);
   SimpleOpenTest(["bin","wc"], vals.fileAttr, <FS_OpenWrite>);
   SimpleOpenTest(["etc", "resolv.conf"], vals.fileAttr, <FS_OpenWrite>));
```

Code 4.115: Tests for VDM++ model sliced at FS_OpenFileDir — Test case at System level

Such as the test at the API level done for the `FS_DeleteFileDir` operation, for this operation all possible returned status values are tested. However the result of this operation is not simply a status value, it can also return a handle to a file. This makes it handy to write a test method that based on the file type can check if the resulting handler is `nil` or not.

```
SimpleOpenTest(path, attr, omode) ==
  let mk_(handle,status) = FS_OpenFileDir(path, attr, omode) in
```

```
  (if (attr.fileType = <RegularFile> and status = <FFS_StatusSuccess>)
   then (AssertTrue(handle <> nil);
         AssertTrue(sys.table(handle).fileOffset = getOpenOffset(sys.fileStore(path),
             omode))
        )
   else AssertTrue(handle = nil);
   return status);
```

Code 4.116: Tests for VDM++ model sliced at FS_OpenFileDir — SimpleOpenTest at the API level

In the test case the return status values are checked according with the expected conditions.

```
RunTest: () ==> ()
RunTest() ==
  (dcl status : FFS_Status;

   status := SimpleOpenTest(["etc","hosts"], vals.fileAttr, <FS_OpenAlways>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(<Root>, vals.dirAttr, <FS_CreateAlways>);
   AssertTrue(status = <FS_ErrorDirectoryNonEmpty>);

   status := SimpleOpenTest(["bin","ls"], vals.fileAttr, <FS_CreateAlways>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(["bin","wc"], vals.fileAttr, <FS_OpenRead>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(["bin"], vals.dirAttr, <FS_OpenRead>);
   AssertTrue(status = <FFS_StatusInvalidParameter>);

   status := SimpleOpenTest(["bin"], vals.dirAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorFileAlreadyExists>);

   status := SimpleOpenTest(["etc","resolv.conf"], vals.fileAttr, <FS_CreateAlways>);
   AssertTrue(status = <FS_ErrorFileStillOpen>);

   status := SimpleOpenTest(["usr"], vals.dirAttr, <FS_OpenRead>);
   AssertTrue(status = <FS_ErrorFileNotFound>);

   status := SimpleOpenTest(["usr"], vals.dirAttr, <FS_OpenWrite>);
   AssertTrue(status = <FS_ErrorFileNotFound>);

   status := SimpleOpenTest(["usr","share"], vals.dirAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorInvalidPath>);

   status := SimpleOpenTest(["bin","ls","somefile"], vals.fileAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorInvalidPath>));
```

Code 4.117: Tests for VDM++ model sliced at FS_OpenFileDir — Test case at the API level

4.2.4 Alloy Model

As the VDM++ specification grows in detail also must do its Alloy counter part, so that model checking can be performed. The new enumerated types `FS_OpenMode` and `FS_AccessMode` get translated to Alloy in the same way as `FileType`.

```
abstract sig FS_OpenMode {}  -- new for open
one sig FS_CreateNew             extends FS_OpenMode {}-- new for open
one sig FS_CreateAlways          extends FS_OpenMode {}-- new for open
one sig FS_OpenRead              extends FS_OpenMode {}-- new for open
one sig FS_OpenWrite             extends FS_OpenMode {}-- new for open
one sig FS_OpenAlways            extends FS_OpenMode {}-- new for open
one sig FS_OpenWriteOnly         extends FS_OpenMode {}-- new for open
one sig FS_CreateAlwaysReadOnly extends FS_OpenMode {}-- new for open
one sig FS_CreateNewReadOnly     extends FS_OpenMode {}-- new for open
```

Code 4.118: Alloy model sliced at FS_OpenFileDir — FS_OpenMode signature

```
abstract sig FS_AccessMode {} -- new for open
one sig  FS_AccessReadOnly  extends FS_AccessMode {} -- new for open
one sig  FS_AccessWriteOnly extends FS_AccessMode {} -- new for open
one sig  FS_AccessReadWrite extends FS_AccessMode {} -- new for open
```

Code 4.119: Alloy model sliced at FS_OpenFileDir — FS_AccessMode signature

The mapping between open and access modes gets translated to Alloy as a predicate with the appropriate logic, so that only valid mode pairs match. This could also be translated by a simple relation in Alloy, although the static nature of this mapping that is equivalent to its original version in VDM++ allows for it to be specified as a predicate.

```
pred fs_open2access_mode_map[omode: FS_OpenMode, amode: FS_AccessMode] { -- new for open
  (omode = FS_CreateNew             and amode = FS_AccessReadWrite) or
  (omode = FS_CreateAlways          and amode = FS_AccessReadWrite) or
  (omode = FS_OpenRead              and amode = FS_AccessReadOnly)  or
  (omode = FS_OpenWrite             and amode = FS_AccessReadWrite) or
  (omode = FS_OpenAlways            and amode = FS_AccessReadWrite) or
  (omode = FS_OpenWriteOnly         and amode = FS_AccessWriteOnly) or
  (omode = FS_CreateAlwaysReadOnly and amode = FS_AccessReadOnly)  or
  (omode = FS_CreateNewReadOnly     and amode = FS_AccessReadOnly)
}
```

Code 4.120: Alloy model sliced at FS_OpenFileDir — fs_open2access_mode_map predicate

Files now have contents and in VDM++ these are modeled as a sequence of `token` values. The translation of `FileContents` to Alloy could be done using sequences although, as in the case of `Path`, the option was to abstract the sequence and establish a relation between elements. The actual `FileContents` signature is declared as abstract, so that no objects of tat type can be created.

```
abstract sig FileContents extends OptionalFileContents {}  -- new for open
```

Code 4.121: Alloy model sliced at FS_OpenFileDir — FileContents signature

Two other signatures `Chunk`, representing some content, and `Nothing`, representing no content, are declared to extend `FileContents`. Chunks of content are related to other `FileContent` el-

ements through the relation `nextChunk`, which is constrained to be a function and acyclic by the `FileContensInvariantVDM` predicate.

```
one sig Nothing extends FileContents {}  -- new for open
```

Code 4.122: Alloy model sliced at FS_OpenFileDir — Nothing signature

```
sig Chunk extends FileContents {  -- new for open
  nextChunk: FileContents
}    -- some contents
```

Code 4.123: Alloy model sliced at FS_OpenFileDir — Chunk signature

```
pred FileContentsInvariantVDM[cont: FileContents] {  -- new for open
  RelCalc/Function[nextChunk,Chunk,FileContents] and
  RelCalc/Acyclic[(RelCalc/id[Chunk]).nextChunk,Chunk]
}
```

Code 4.124: Alloy model sliced at FS_OpenFileDir — FileContentsInvariantVDM predicate

The translation of `FileContens` is very similar to the translation of `Path`, although with a slight difference on the fact that the termination element of the `nextChunk` relation does not belong to the relation's domain. The signature `File` also need to be updated to include the respective contents, and a new invariant. In the VDM++ model file contents are optional and this is declared using the parametric optional type instantiated with the type `FileContents`.

```
sig File {
  info     : FS_FileDirInfo,
  contents : OptionalFileContents   -- new for open
}
```

Code 4.125: Alloy model sliced at FS_OpenFileDir — File signature

```
pred FileInvariantVDM[f: File]{
  FS_FileDirInfoInvariantVDM[f.info]
  OptionalFileContentsInvariantVDM[f.contents] and   -- new for open
  all file: f {
    file.contents not in NilFileContents
    => let fileContents = file.contents.(^nextChunk) |
       no (fileContents & (File.contents.(^nextChunk) - fileContents))
  }                   -- new for open
  FileInvariant[f]   -- new for open
}
```

Code 4.126: Alloy model sliced at FS_OpenFileDir — FileInvariantVDM predicate

```
pred FileInvariant[f: File] {   --- new for open
  all file: f {
    (file.info.attributes.fileType in Directory and
     file.contents in NilFileContents)
    or
    (file.info.attributes.fileType in RegularFile and
     file.contents in FileContents)
  }
}
```

Code 4.127: Alloy model sliced at FS_OpenFileDir — FileInvariant predicate

Mathematically the sequence and optional types are expressed through the isomorphisms (4.1) and (4.2),

$$X^* \cong 1 + X \times X^* \tag{4.1}$$

and

$$[X] \cong 1 + X \tag{4.2}$$

respectively. The fact that both data types are defined as a disjoint co-product of singleton value type 1 and something else, makes the translation to Alloy similar in its structure. So the Alloy signature `OptionalFileContents` is declared abstract, just such as `FileContents`, and extended by two other signatures: the `NilFileContents` denoting the mathematical type 1 (just such as `Nothing`); and `FileContents`.

```
abstract sig OptionalFileContents {} -- new for open
```

Code 4.128: Alloy model sliced at FS_OpenFileDir — OptionalFileContents signature

```
one sig NilFileContents extends OptionalFileContents {} -- new for open
```

Code 4.129: Alloy model sliced at FS_OpenFileDir — NilFileContents signature

```
pred OptionalFileContentsInvariantVDM[ofc: OptionalFileContents] { -- new for open
  FileContentsInvariantVDM[(ofc & FileContents)]
}
```

Code 4.130: Alloy model sliced at FS_OpenFileDir — OptionalFileContentsInvariantVDM predicate

The `OptionalFileContentsInvariantVDM` predicate enforces the subtype `FileContents` invariant. Another updated structure is `FS_OpenFileInfo`, that for this operation needs to keeps track of more information, namely the offset with the file and the granted access to the file. Due to the abstraction of file contents as a relation, to deal with the file offset is necessary to make an adaptation, instead of simply using a number.

```
sig FS_OpenFileInfo {
  fileOffset:  FileContents,  -- new for open
  accessMode:  FS_AccessMode, -- new for open
  path      :  Path
}
```

Code 4.131: Alloy model sliced at FS_OpenFileDir — FS_OpenFileInfo signature

Instead of an index in a sequence, the `fileOffset` field in `FS_OpenFileInfo` is the actual referenced content. The invariant for this data type just enforces subtype's invariants.

```
pred FS_OpenFileInfoInvariantVDM[ofi: FS_OpenFileInfo]{
  FileContentsInvariantVDM[ofi.fileOffset]       and   -- new for open
  PathInvariantVDM[ofi.path]
}
```

Code 4.132: Alloy model sliced at FS_OpenFileDir — FS_OpenFileInfoInvariantVDM predicate

The mappings of the two components of the system, namely `FileStore` and `OpenFilestable`, have to be declared injective. Until now, any two files pointed by the same path were interpreted as two different files, although with the same value. The option was taken to reduce the amount of objects that the solver would need to instantiate, although it was known that changes to one file's info could lead to side effects on other files. With the inclusion of file contents in the specification it becomes harder to sustain such option, as file contents tent do change allot. The dynamic nature of file contents makes the injectiveness of the file store mapping a question of consistency, because changes made to some file's content should not affect other files.

```
pred FileStoreInvariantVDM[fs: FileStore] {
  RelCalc/Simple[fs.map, File]           and
  RelCalc/Injective[fs.map, Path]        and   -- new for open
  PathInvariantVDM[RelCalc/dom[fs.map]] and
  FileInvariantVDM[RelCalc/rng[fs.map]] and
  FileStoreInvariant[fs]
}
```

Code 4.133: Alloy model sliced at FS_OpenFileDir — FS_FileStoreInvariantVDM predicate

The same happens with the `OpneFilesTable` signature, because `FS_OpenFileInfo` now can point to some content of a file.

```
pred OpenFilesTableInvariantVDM[table: OpenFilesTable] {
  RelCalc/Simple[table.map,FS_OpenFileInfo]           and
  RelCalc/Injective[table.map, FS_FileHandle]         and
  FS_OpenFileInfoInvariantVDM[RelCalc/rng[table.map]]
}
```

Code 4.134: Alloy model sliced at FS_OpenFileDir — FS_OpenFilesTableInvariantVDM predicate

The system structure remained the same, although its invariant has an extra clause stating that there can only be files referenced in the open files table.

```
pred OpenFilesTableInvariantVDM[table: OpenFilesTable] {
```

Code 4.135: Alloy model sliced at FS_OpenFileDir — SystemInvariant predicate

The boolean functions `checkOpenMode` and `mustDeleteFirst` are translated to predicates with the same name, and logic respectively.

```
pred checkOpenMode[sys  : System,
                   path : Path,
                   omode: FS_OpenMode] {  -- new for open
  not (isCreateNew[omode]    and isElemFileStore[path,sys.fileStore])     and
  not (isCreateAlways[omode] and isElemTablePath[path,sys.table])         and
  not (isOpen[omode]         and not isElemFileStore[path,sys.fileStore])
}
```

Code 4.136: Alloy model sliced at FS_OpenFileDir — checkOpenMode predicate

```
pred mustDeleteFirst[fs: FileStore, path: Path, omode: FS_OpenMode] {   -- new for open
  isCreateAlways[omode] and isElemFileStore[path,fs]
}
```

Code 4.137: Alloy model sliced at FS_OpenFileDir — mustDeleteFirst predicate

The `FS_OpenFileDir_Main` function is translated to a predicate, using the same kind of conditional logic as in `FS_DeleteFileDir_Main`.

```
pred FS_OpenFileDir_Main[sys,sys'  : System,
                         full_path : Path,
                         attributes: Attributes,
                         omode     : FS_OpenMode,
                         handle    : OptionalFileHandle,
                         status    : FFS_Status         ] {  -- new for open
  (pre_FS_OpenFileDir_System[sys,full_path,attributes,omode] and
   checkOpenMode[sys,full_path,omode])
  =>   (mustDeleteFirst[sys.fileStore,full_path,omode] and
        (full_path = Root => attributes.fileType = Directory))
       =>   (some dstatus: FFS_Status, dsys: System {
               FS_DeleteFileDir_Main[sys,dsys,full_path,dstatus] and
               dstatus = FFS_StatusSuccess
               =>   (FS_OpenFileDir_System[dsys,sys',full_path,attributes,omode,handle]
                   and
                     status = FFS_StatusSuccess)
               else (sys'   = sys          and
                     handle = NilFileHandle and
                     status = dstatus)
             })
       else (FS_OpenFileDir_System[sys,sys',full_path,attributes,omode,handle] and
             status = FFS_StatusSuccess)
  else (sys'   = sys           and
        handle = NilFileHandle and
        FS_OpenFileDir_Exception[sys,full_path,omode,status])
}
```

Code 4.138: Alloy model sliced at FS_OpenFileDir — FS_OpenFileDir_Main predicate

However, the predicate introduces an existential quantifier to pipe the result of removing a file on the original system in to the call to the `FS_OpenFileDir_System`, which opens and/or creates files. The quantification is done using the `some` multiplicity factor, that allows for one or more elements of a relation to match the given criteria. If the multiplicity factor used was `one` this would mean that the relation would have one and only one possible element, thus constraining more than needed.

```
pred FS_OpenFileDir_System[sys,sys' : System,
                           full_path: Path,
                           attr     : Attributes,
                           omode    : FS_OpenMode,
                           handle   : OptionalFileHandle] {  -- new for open
  FS_OpenFileDir_FileStore[sys.fileStore,sys'.fileStore,full_path,attr] and
  let fileType = full_path.(sys'.fileStore.map).info.attributes.fileType {
    FS_OpenFileDir_Table[sys.table,sys'.table,full_path,omode,fileType,handle]
  }
}
```

Code 4.139: Alloy model sliced at FS_OpenFileDir — FS_OpenFileDir_System predicate

```
pred pre_FS_OpenFileDir_System[sys      : System,
                               full_path: Path,
                               attr     : Attributes,
                               omode    : FS_OpenMode] {  -- new for open
  pre_FS_OpenFileDir_FileStore[sys.fileStore,full_path,attr]
```

```
}
```

Code 4.140: Alloy model sliced at FS_OpenFileDir — pre_FS_OpenFileDir_System predicate

The `getOpenOffset` original function in the VDM++ model was to return a index to a position in a sequence of contents within a file, although, as mentioned above, that particular string was abstracted through different signatures and a binary relation, and referencing bits of content is done by including the desired object in the relation `fileOffset`. So the predicate `getOpenOffset` will logically specify just that.

```
pred getOpenOffset[file: File, omode: FS_OpenMode, contents: FileContents] {  -- new for
     open
  (omode in FS_OpenWrite + FS_OpenAlways and contents = getLastChunk[file]) or
  (contents = Nothing)
}
```

Code 4.141: Alloy model sliced at FS_OpenFileDir — getOpenOffset predicate

In the above predicate definition there is a call to an Alloy function that given a file returns the last chunk of that file's contents. This function could be abstracted as a predicate, with input and output parameters, as done so far with other functions. In this case the purpose is to illustrate a translation of a VDM++ function to an Alloy function.

```
fun getLastChunk[file: File] : FileContents {  -- new for open
  file.contents in Nothing
  =>   Nothing
  else RelCalc/dom[(file.contents.(*nextChunk)->Nothing) & nextChunk]
}
```

Code 4.142: Alloy model sliced at FS_OpenFileDir — getLastChunk function

The `FS_OpenFileDir_FileStore` predicate is the translation of the VDM++ function that has the same name. This translation introduces another existential quantifier, although more restrictive than the last. Here the goal is to create just one new connection between path and file.

```
pred FS_OpenFileDir_FileStore[fs,fs'   : FileStore,
                              full_path: Path,
                              attr     : Attributes] {  -- new for open
  not isElemFileStore[full_path,fs]
  =>   (one file: File {
          fs'.map = fs.map + (full_path -> file)                 and
          file.info.attributes = attr                            and
          (attr.fileType in Directory => file.contents in NilFileContents) and
          (attr.fileType in RegularFile => file.contents in FileContents) and
          not isElemFileStore[file,fs']
       })
  else fs'.map = fs.map
}
```

Code 4.143: Alloy model sliced at FS_OpenFileDir — FS_OpenFileDir_FileStore predicate

```
pred pre_FS_OpenFileDir_FileStore[fs        : FileStore,
                                  full_path: Path,
                                  attr     : Attributes] {  -- new for open
  (full_path = Root and
```

```
   attr.fileType = Directory)
  or
  (isElemFileStore[full_path.dirName,fs] and
   isDirectory[(fs.map[full_path.dirName]).info])
}
```

Code 4.144: Alloy model sliced at FS_OpenFileDir — pre_FS_OpenFileDir_FileStore predicate

The `FS_OpenFileDir_Table` function of the VDM++ model gets translated to an Alloy predicate, that in the case of a regular file creates just one new element in the binary relation between file handlers and open file info called `OpenFilesTable.map`.

```
pred FS_OpenFileDir_Table[table,table': OpenFilesTable,
                          full_path   : Path,
                          omode       : FS_OpenMode,
                          fileType    : FileType,
                          handle      : OptionalFileHandle] {  -- new for open
  fileType in Directory
  =>   (table'.map = table.map and
        handle = NilFileHandle)
  else one ofi: FS_OpenFileInfo {
         not isElemTableHandle[handle,table]            and
         not isElemTable[ofi,table]                     and
         ofi.fileOffset = Nothing                       and
         ofi.path = full_path                           and
         fs_open2access_mode_map[omode,ofi.accessMode]  and
         table'.map = table.map + (handle -> ofi)
       }
}
```

Code 4.145: Alloy model sliced at FS_OpenFileDir — FS_OpenFileDir_Table predicate

For last comes the translation of both the function `FS_OpenFileDir_Exception`.

```
pred FS_OpenFileDir_Exception[sys      : System,
                              full_path: Path,
                              omode    : FS_OpenMode,
                              status   : FFS_Status] {  -- new for open
  (isCreateNew[omode] and isElemFileStore[full_path,sys.fileStore])
  =>   status = FS_ErrorFileAlreadyExists
  else (isCreateAlways[omode] and isElemTablePath[full_path,sys.table])
       =>   status = FS_ErrorFileStillOpen
       else (isOpen[omode] and not isElemFileStore[full_path,sys.fileStore])
            =>   status = FS_ErrorFileNotFound
            else (isOpen[omode] and isElemFileStore[full_path, sys.fileStore]
                                and not isRegularFile[sys.fileStore.map[full_path].info
                                    ])
                 => status = FFS_StatusInvalidParameter
                 else not (full_path in Root or
                           isElemFileStore[full_path.dirName,sys.fileStore])
                      =>   status = FS_ErrorInvalidPath
                      else not (full_path in Root or
                                isDirectory[(sys.fileStore.map[full_path.dirName]).info
                                    ])
                           =>   status = FS_ErrorInvalidPath
```

Code 4.146: Alloy model sliced at FS_OpenFileDir — FS_OpenFileDir_Exception predicate

4.2.5 Model checking the operation with the Alloy Analyzer

Model checking the `FS_OpenFileDir` operation is much alike model checking the previous operation. Assertions are made about the predicates that compose the operation. Predicates *_FileStore*, *_Table* and *_System* are checked for satisfiability.

```
assert Open_FileStore {
  all fs,fs': FileStore, full_path: Path, attr: Attributes {
   FileStoreInvariantVDM[fs]                       and
   PathInvariantVDM[full_path]                     and
   pre_FS_OpenFileDir_FileStore[fs,full_path,attr] and
   FS_OpenFileDir_FileStore[fs,fs',full_path,attr]
   => FileStoreInvariantVDM[fs']         and
      full_path in RelCalc/dom[fs'.map]  and
      (not isElemFileStore[full_path,fs]
       => full_path.(fs'.map).info.attributes = attr)
  }
}
Check_Open_FileStore: check Open_FileStore
                      for 7 but 0 System,
                                0 OpenFilesTable
```

Code 4.147: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_FileStore

```
assert Open_Table {
  all table,table': OpenFilesTable, full_path: Path, omode: FS_OpenMode,
    handle: OptionalFileHandle, ft: FileType {
    OpenFilesTableInvariantVDM[table]                             and
    PathInvariantVDM[full_path]                                   and
    FS_OpenFileDir_Table[table,table',full_path,omode,ft,handle]
    => ft in RegularFile
       =>   full_path = handle.(table'.map).path                          and
            RelCalc/dom[table'.map] = RelCalc/dom[table.map] + handle     and
            OpenFilesTableInvariantVDM[table']                            and
            fs_open2access_mode_map[omode,handle.(table'.map).accessMode]
       else table'.map = table.map
  }
}
Check_Open_Table: check Open_Table
                  for 7 but 0 System,
                            0 FileStore,
                            2 OpenFilesTable
```

Code 4.148: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Table

```
assert Open_System {
  all sys,sys': System, full_path: Path, attr: Attributes,
      omode: FS_OpenMode, amode: FS_AccessMode, handle: OptionalFileHandle {
    fs_open2access_mode_map[omode,amode]                         and
    SystemInvariantVDM[sys]                                      and
    PathInvariantVDM[full_path]                                  and
    pre_FS_OpenFileDir_System[sys,full_path,attr,omode]          and
    FS_OpenFileDir_System[sys,sys',full_path,attr,omode,handle]
    => SystemInvariantVDM[sys']                                        and
       full_path in RelCalc/dom[sys'.fileStore.map]                    and
       (not isElemFileStore[full_path,sys.fileStore]
        => full_path.(sys'.fileStore.map).info.attributes = attr)      and
```

```
       (isDirectory[full_path.(sys'.fileStore.map).info]
        =>   (full_path not in FS_FileHandle.(sys'.table.map).path and
              handle in NilFileHandle)
        else (full_path = handle.(sys'.table.map).path) and
              handle.(sys'.table.map).accessMode = amode)
  }
}
Check_Open_System: check Open_System
                   for 7 but 2 System
```

Code 4.149: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_System

The *_Table* predicate is also checked for directory opening, and although at the `System` level the same property can be checked it is not needed, because the property is part of the `SystemInvariant` predicate.

```
assert Open_Table_Directories {
  all table,table': OpenFilesTable, full_path: Path,
      omode: FS_OpenMode, handle: OptionalFileHandle, ft: FileType {
    ft = Directory                                              and
    OpenFilesTableInvariantVDM[table]                           and
    PathInvariantVDM[full_path]                                 and
    FS_OpenFileDir_Table[table,table',full_path,omode,ft,handle]
    => full_path not in handle.(table'.map).path
  }
}
Check_Open_Table_Directories: check Open_Table_Directories
                              for 7 but 0 System,
                                        0 FileStore,
                                        2 OpenFilesTable
```

Code 4.150: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Table for directory opening

The *_Exception* predicate is checked both to provide the adequate status values for each expected condition, and that the unknown status is not a valid result outside the condition for the call for error handling in the *_Main* predicate. The operation informal specification dictates the error conditions for some possible open modes. Although the model checking of this *_Exception* predicate is done according to specific error status all the open modes related error situations are covered.

```
assert Open_Exception_FileAlreadyExists {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]             and
    PathInvariantVDM[path]              and
    isElemFileStore[path,sys.fileStore] and
    isCreateNew[omode]                  and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileAlreadyExists
  }
}
Check_Open_Exception_FileAlreadyExists: check Open_Exception_FileAlreadyExists
                                        for 7
```

Code 4.151: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FS_FileAlreadyExists

```
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]          and
    PathInvariantVDM[path]           and
    isElemTablePath[path,sys.table]  and
    isCreateAlways[omode]            and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileStillOpen
  }
}
Check_Open_Exception_FileStillOpen: check Open_Exception_FileStillOpen
                                    for 7 but 1 System
```

Code 4.152: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FS_ErrorFileStillOpen

```
    SystemInvariantVDM[sys]                  and
    PathInvariantVDM[path]                   and
    not isElemFileStore[path,sys.fileStore]  and
    isOpen[omode]                            and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileNotFound
  }
}
Check_Open_Exception_FileNotFound: check Open_Exception_FileNotFound
                                   for 7
```

Code 4.153: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FS_ErrorFileNotFound

```
    PathInvariantVDM[path]                           and
    isOpen[omode]                                    and
    isElemFileStore[path, sys.fileStore]             and
    not isRegularFile[sys.fileStore.map[path].info]  and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FFS_StatusInvalidParameter
  }
}
Check_Open_Exception_InvalidParameter: check Open_Exception_InvalidParameter
                                       for 7
```

Code 4.154: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FS_StatusInvalidParameter

```
    (not path in Root and
     (not isElemFileStore[path.dirName,sys.fileStore] or
      not isDirectory[(sys.fileStore.map[path.dirName]).info])) and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorInvalidPath
  }
}
Check_Open_Exception_InvalidPath: check Open_Exception_InvalidPath
                                  for 7
```

Code 4.155: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FS_ErrorInvalidPath

```
assert Open_Exception_StatusUnknown {
  all sys: System, path: Path, omode: FS_OpenMode,
      attr: Attributes, status: FFS_Status {
    SystemInvariantVDM[sys]                          and
    PathInvariantVDM[path]                           and
    not (pre_FS_OpenFileDir_System[sys,path,attr,omode] and
         checkOpenMode[sys,path,omode])  and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => not status in FFS_StatusUnknown
  }
}
Check_Open_Exception_StatusUnknown: check Open_Exception_StatusUnknown
```

Code 4.156: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Exception for FFS_StatusUnknown

At the top most *_Main* predicate the model is checked for the success and error cases generally.

```
    checkOpenMode[sys,full_path,omode]                  and
    pre_FS_OpenFileDir_System[sys,full_path,attr,omode] and
    FS_OpenFileDir_Main[sys,sys',full_path,attr,omode,handle,status]
    => SystemInvariantVDM[sys'] and
       status = FFS_StatusSuccess
  }
}
Check_Open_Main_Success: check Open_Main_Success
                         for 7
```

Code 4.157: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Main success cases

```
    PathInvariantVDM[full_path]         and
    not(pre_FS_OpenFileDir_System[sys,full_path,attr,omode] and
        checkOpenMode[sys,full_path,omode]) and
    FS_OpenFileDir_Main[sys,sys',full_path,attr,omode,handle,status]
    => SystemInvariantVDM[sys'] and
       status not = FFS_StatusSuccess
  }
}
Check_Open_Main_Error: check Open_Main_Error
                       for 7
```

Code 4.158: Alloy model sliced at FS_OpenFileDir — Check FS_OpenFileDir_Main error cases

4.2.6 Model Checking VDM Proof Obligations with the Alloy Analyser

The current VDM++ specification generates much more POs than the previous, in this case there are forty POs to model check. Some of the POs generated for this operation were already checked in the previous operation and, although they are checked again, will be omitted.

```
Integrity property #1 :
In function FileSystemLayerBase newFileHandle, file: FileSystemLayerBase.vpp l. 323 c.
    38: function application from max
-------------------------------------------------------------------------------
(forall handles : set of FS_FileHandle &
```

```
not (card (handles) = 0) =>
 FileSystemLayerBase`pre_max(handles))
```

Code 4.159: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 2

```
assert po2 {
  all handles: set FS_FileHandle {
    not (#handles = 0) => #handles > 0
  }
}
CheckPO2: check po2 for 7
```

Code 4.160: Alloy model sliced at FS_OpenFileDir — Proof Obligation 2

PO2 is generated from function `newFileHandler` because it uses the partial function `max`, and it should do it within the boundaries set by `max`'s pre-condition. Although `max` was not needed in the Alloy model, because file handles are more abstract than their VDM++ counterpart, it is possible to model check the previous PO since `pre_max` is known and quite simple. However the option of not modeling the max relation in Alloy makes it impossible to check POs that use the `max` function. Specifying the max relation would not be any different that specifying the `dirName` relation, which is a partial order on paths. Some other POs related with paths and file names and the `is_` operator for data types with no invariant will be also skipped.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 213 c. 3: invariants from FileStore
-------------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
(full_path = <Root> and
 attributes.fileType = <Directory>) or
 ((let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) and
 isDirectory(fileStore(parent).info))) =>
 FileSystemLayerOperations`inv_FileStore((if not (isElemFileStore(full_path, fileStore))
      then
(let content = emptyFileContents(attributes.fileType),
     newFile = mk_File(newFileDirInfo(attributes),content)
in
    fileStore munion {full_path |-> newFile})
else
fileStore)))
```

Code 4.161: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 10

```
assert po10 {
  all fileStore, fileStore': FileStore,
      full_path: Path, attr: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => (full_path in Root and attr.fileType in Directory) or
       (let parent = full_path.dirName |
        isElemFileStore[parent, fileStore] and
```

```
        isDirectory[fileStore.map[parent].info])
       => (not isElemFileStore[full_path, fileStore]
           =>   (one newFile: File {
                  fileStore'.map = fileStore.map + (full_path -> newFile)         and
                  newFile.info.attributes = attr                                  and
                  (attr.fileType in Directory => newFile.contents in NilFileContents)
                      and
                  (attr.fileType in RegularFile => newFile.contents in FileContents)
                      and
                  not isElemFileStore[newFile,fileStore']
                 })
                => FileStoreInvariantVDM[fileStore']
           else FileStoreInvariantVDM[fileStore])
  }
}
CheckPO10: check po10 for 7
```

Code 4.162: Alloy model sliced at FS_OpenFileDir — Proof Obligation 10

PO10 is generated from function `FS_OpenFileDir_FileStore` because it returns a `FileStore`, and the resulting value should preserve the data type invariant.

```
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 216 c. 18: compatible maps
-------------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
(full_path = <Root> and
 attributes.fileType = <Directory>) or
 ((let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) and
 isDirectory(fileStore(parent).info))) =>
 not (isElemFileStore(full_path, fileStore)) =>
 (let content = emptyFileContents(attributes.fileType),
      newFile = mk_File(newFileDirInfo(attributes),content)
in
    (forall id_9 in set dom (fileStore), id_10 in set dom ({full_path |-> newFile}) &
id_9 = id_10 =>
 fileStore(id_9) = {full_path |-> newFile}(id_10))))
```

Code 4.163: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 11

```
assert po11 {
  all fileStore: FileStore,
      full_path: Path, attr: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => (full_path in Root and attr.fileType in Directory) or
       (let parent = full_path.dirName |
        isElemFileStore[parent, fileStore] and
        isDirectory[fileStore.map[parent].info])
       => not isElemFileStore[full_path, fileStore]
          =>  (some newFile: File {
                newFile.info.attributes = attr                                      and
                (attr.fileType in Directory => newFile.contents in NilFileContents) and
```

```
                (attr.fileType in RegularFile => newFile.contents in FileContents)
                => all id_9,id_10: Path {
                     PathInvariantVDM[id_9]  and
                     PathInvariantVDM[id_10]
                     => id_9  in RelCalc/dom[fileStore.map] and
                        id_10 in RelCalc/dom[(full_path->newFile)]
                        => id_9 = id_10
                           => fileStore.map[id_9] = (full_path->newFile)[id_10]
                   }
              })
  }
}
CheckPO11: check po11 for 7
```

Code 4.164: Alloy model sliced at FS_OpenFileDir — Proof Obligation 11

PO11 is generated from function `FS_OpenFileDir_FileStore`, and it states that every file store, resulting from the function, should preserve the simplicity property after adding a new entry to the mapping.

```
Integrity property #3 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 220 c. 65: map application
----------------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
not ((full_path = <Root> and
 attributes.fileType = <Directory>)) =>
 (let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) =>
 parent in set dom (fileStore)))
```

Code 4.165: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 12

```
assert po12 {
  all fileStore: FileStore, full_path: Path, attributes: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => not (full_path in Root and attributes.fileType in Directory)
       => let parent = full_path.dirName {
            isElemFileStore[parent, fileStore]
            => parent in RelCalc/dom[fileStore.map]
          }
  }
}
CheckPO12: check po12 for 7
```

Code 4.166: Alloy model sliced at FS_OpenFileDir — Proof Obligation 12

PO12 targets the same function as the previous, although in this case the focus is on the application of the file store parameter to a the parent path of the full path parameter. PO13 and PO14 are generated form the `FS_OpenFileDir_Exception` and also target partial mapping applications.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_Exception , file:
    FileSystemLayerOperations.vpp l. 260 c. 41: map application
--------------------------------------------------------------------------------
(forall sys : System, full_path : Path, omode : FS_OpenMode &
not (isCreateNew(omode) and
 isElemFileStore(full_path, sys.fileStore)) =>
 not (isCreateAlways(omode) and
 isElemTablePath(full_path, sys.table)) =>
 not (isOpen(omode) and
 not (isElemFileStore(full_path, sys.fileStore))) =>
 isOpen(omode) and
 isElemFileStore(full_path, sys.fileStore) =>
 full_path in set dom (sys.fileStore))
```

Code 4.167: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 13

```
assert po13 {
  all sys: System, full_path: Path, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => not (isCreateNew[omode] and isElemFileStore[full_path, sys.fileStore])
       => not (isCreateAlways[omode] and isElemTablePath[full_path, sys.table])
          => not (isOpen[omode] and not isElemFileStore[full_path, sys.fileStore])
             => isOpen[omode] and isElemFileStore[full_path, sys.fileStore]
                => full_path in RelCalc/dom[sys.fileStore.map]
  }
}
CheckPO13: check po13 for 7
```

Code 4.168: Alloy model sliced at FS_OpenFileDir — Proof Obligation 13

PO14 was omitted as it is very similar to PO13. PO20 and PO22 are generated from function `FS_OpenFileDir_System` as are all POs until PO30. The first states that the `System` instance resulting from the function must preserve the data type's invariant.

```
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_System , file:
    FileSystemLayerOperations.vpp l. 164 c. 3e: invariants from System
--------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 FileSystemLayerOperations`inv_System((let fileStore' = FS_OpenFileDir_FileStore(sys.
     fileStore, full_path, attr), mk_(table,handle) = FS_OpenFileDir_Table(sys.table,
     full_path, omode, attr.fileType), offset = getOpenOffset(fileStore'(full_path),
     omode), table' = table ++ (if handle <> nil and
 isElemTableHandle(handle, table) then
{handle |-> mu(table(handle),fileOffset|->offset)}
else
{|->})
in
    mk_(mk_System(table',fileStore'),handle)).#1))
```

Code 4.169: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 20

```
assert po20 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle, offset: FileContents,
              table',t': OpenFilesTable, ofi: FS_OpenFileInfo {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
                and
            getOpenOffset[fileStore'.map[full_path], omode, offset] and
            (handle not in NilFileHandle and isElemTableHandle[handle, t]
             =>   (ofi.accessMode = t.map[handle].accessMode and
                   ofi.path = t.map[handle].path              and
                   ofi.fileOffset = offset
                   => t'.map = (handle->ofi))
             else no t'.map)
            table'.map = (t.map - (RelCalc/dom[t'.map]->FS_OpenFileInfo)) + t'.map
            => all sys': System {
                 sys'.table = table'          and
                 sys'.fileStore = fileStore'
                 => SystemInvariantVDM[sys']
               }
          }
  }
}
CheckPO20: check po20 for 7
```

Code 4.170: Alloy model sliced at FS_OpenFileDir — Proof Obligation 20

The latter targets the partial function `FS_OpenFileDir_FileStore` application, stating that is must be done within the boundaries defined in its precondition.

```
Integrity property #3 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 164 c. 51: function application from
    FS_OpenFileDir_FileStore
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 FileSystemLayerOperations`pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr))
```

Code 4.171: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 22

```
assert po22 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
  }
}
CheckPO22: check po22 for 7
```

Code 4.172: Alloy model sliced at FS_OpenFileDir — Proof Obligation 22

PO23 states that the newly created file should preserve the `File` data type invariant.

```
Integrity property #4 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 166 c. 48: invariants from File
-----------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType)
in
    FileSystemLayerOperations`inv_File(fileStore'(full_path))))
```

Code 4.173: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 23

```
assert po23 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => FileInvariantVDM[fileStore'.map[full_path]]
          }
  }
}
CheckPO23: check po23 for 7
```

Code 4.174: Alloy model sliced at FS_OpenFileDir — Proof Obligation 23

When opening a file, `FS_OpenFileDir_System` must call the `getOpenOffset` function to calculate the appropriate file offset to be set in the open file info stored in the open files table. To do so, it needs to access the newly create file store to obtain the file. PO24 targets the partial mapping application of the newly created file store to the full path parameter.

```
Integrity property #5 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 166 c. 48: map application
-----------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType)
in
    full_path in set dom (fileStore')))
```

Code 4.175: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 24

```
assert po24 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => full_path in RelCalc/dom[fileStore'.map]
          }
  }
}
CheckPO24: check po24 for 7
```

Code 4.176: Alloy model sliced at FS_OpenFileDir — Proof Obligation 24

PO28 follows from the same reason as PO24, and targets the partial mapping application of the open files table to the newly created file handler.

```
Integrity property #9 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 168 c. 58: map application
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset = getOpenOffset(fileStore'(full_path), omode)
in
    handle <> nil and
 isElemTableHandle(handle, table) =>
 handle in set dom (table)))
```

Code 4.177: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 28

```
assert po28 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => handle not in NilFileHandle and isElemTableHandle[handle, t]
               => handle in RelCalc/dom[t.map]
          }
  }
}
CheckPO28: check po28 for 7
```

Code 4.178: Alloy model sliced at FS_OpenFileDir — Proof Obligation 28

PO30 states that the file store resulting from the application of `FS_OpenFileDir_FileStore` must preserve its type invariant.

```
Integrity property #11 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 170 c. 25: invariants from FileSystemLayerBase`
    FileStore
----------------------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset = getOpenOffset(fileStore'(full_path), omode),
    table' = table ++ (if handle <> nil and isElemTableHandle(handle, table)
                       then {handle |-> mu(table(handle),fileOffset|->offset)}
                         else {|->})
  in
    FileSystemLayerBase`inv_FileStore(fileStore')))
```

Code 4.179: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 30

```
assert po30 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle, offset: FileContents,
              table',t': OpenFilesTable, ofi: FS_OpenFileInfo {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
                and
            getOpenOffset[fileStore'.map[full_path], omode, offset] and
            (handle not in NilFileHandle and isElemTableHandle[handle, t]
             =>   (ofi.accessMode = t.map[handle].accessMode and
                   ofi.path = t.map[handle].path             and
                   ofi.fileOffset = offset
                   => t'.map = (handle->ofi))
             else no t'.map)
            table'.map = (t.map - (RelCalc/dom[t'.map]->FS_OpenFileInfo)) + t'.map
            => FileStoreInvariantVDM[fileStore']
          }
  }
}
CheckPO30: check po30 for 7
```

Code 4.180: Alloy model sliced at FS_OpenFileDir — Proof Obligation 30

PO33 is generated from `FS_OpenFileDir_Table` function, due to the application of the `fs_open2`

```
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_Table , file:
    FileSystemLayerOperations.vpp l. 235 c. 44: map application
------------------------------------------------------------------------------------
(forall table : OpenFilesTable , full_path : Path , omode : FS_OpenMode , fileType :
    FileType &
not (fileType = <Directory>) =>
 omode in set dom (fs_open2access_mode_map))
```

Code 4.181: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 33

```
assert po33 {
  all omode: FS_OpenMode , fileType: FileType {
    not (fileType in Directory)
    => some amode: FS_AccessMode {
         fs_open2access_mode_map[omode,amode]
       }
  }
}
CheckPO33: check po33 for 7
// removed table  and full_path because are not used
```

Code 4.182: Alloy model sliced at FS_OpenFileDir — Proof Obligation 33

In the above PO the two universally quantified free variables `table` and `full_path` were removed from the Alloy assertion. PO34 is also generated from `FS_OpenFileDir_Table`, because of the preservation of the mapping simplicity in the resulting open files table.

```
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_Table , file:
    FileSystemLayerOperations.vpp l. 238 c. 18: compatible maps
------------------------------------------------------------------------------------
(forall table : OpenFilesTable , full_path : Path , omode : FS_OpenMode , fileType :
    FileType &
not (fileType = <Directory>) =>
 (let amode = fs_open2access_mode_map(omode),
      ofi = mk_FS_OpenFileInfo(1,amode,full_path),
    handle = newFileHandle(dom (table))
  in
    (forall id_11 in set dom (table), id_12 in set dom ({handle |-> ofi}) &
       id_11 = id_12 => table(id_11) = {handle |-> ofi}(id_12))))
```

Code 4.183: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 34

```
assert po34 {
  all table: OpenFilesTable , full_path: Path , omode: FS_OpenMode , fileType: FileType {
    not (fileType in Directory)
    => all amode: FS_AccessMode , ofi: FS_OpenFileInfo ,
           handle: FS_FileHandle {
         FS_OpenFileInfoInvariantVDM[ofi]
         => fs_open2access_mode_map[omode,amode] and
            ofi.accessMode = amode               and
            ofi.path = full_path                 and
            handle not in RelCalc/dom[table.map]
```

```
            => all id_11, id_12: FS_FileHandle {
                id_11 in RelCalc/dom[table.map]     and
                id_12 in RelCalc/dom[(handle->ofi)]
                => id_11 = id_12 => table.map[id_11] = (handle->ofi)[id_12]
              }
      }
  }
}
CheckPO34: check po34 for 7
```

Code 4.184: Alloy model sliced at FS_OpenFileDir — Proof Obligation 34

PO36 is generated from the `FS_OpenFileDir_Main`, and states that the `FS_OpenFileDir_System` partial function should only be applied within the boundaries defined by the corresponding precondition.

```
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 111 c. 52: function application from
    FS_OpenFileDir_System
------------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attributes : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
 checkOpenMode(sys, full_path, omode) =>
 mustDeleteFirst(sys.fileStore, full_path, omode) and
 (full_path = <Root> =>
 attributes.fileType = <Directory>) =>
 (let mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path)
in
    not (status <> <FFS_StatusSuccess>) =>
 FileSystemLayerOperations`pre_FS_OpenFileDir_System(sys', full_path, attributes, omode)
     ))
```

Code 4.185: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 36

```
assert po36 {
  all sys: System, full_path: Path, attributes: Attributes,
      omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_System[sys, full_path, attributes, omode] and
       checkOpenMode[sys, full_path, omode]
       => mustDeleteFirst[sys.fileStore, full_path, omode] and
          (full_path in Root => attributes.fileType in Directory)
          => all sys': System, status: FFS_Status {
               status not in FFS_StatusSuccess
               => pre_FS_OpenFileDir_System[sys', full_path, attributes, omode]
             }
  }
}
CheckPO36: check po36 for 7
```

Code 4.186: Alloy model sliced at FS_OpenFileDir — Proof Obligation 36

PO40 is generated from the `checkOpenMode` function, because in its definition there is an application of a path to the a file store, and this application should only be performed if the path is in the file store mapping.

```
Integrity property #1 :
In function FileSystemLayerOperations checkOpenMode , file: FileSystemLayerOperations.vpp
     l. 148 c. 36: map application
-------------------------------------------------------------------------------------------
(forall sys : System, path : Path, omode : FS_OpenMode &
not ((isCreateNew(omode) and
 isElemFileStore(path, sys.fileStore))) and
 not ((isCreateAlways(omode) and
 isElemTablePath(path, sys.table))) and
 not ((isOpen(omode) and
 not (isElemFileStore(path, sys.fileStore)))) =>
 (isOpen(omode) or
 omode = <FS_OpenAlways>) and
 isElemFileStore(path, sys.fileStore) =>
 path in set dom (sys.fileStore))
```

Code 4.187: VDM++ model sliced at FS_OpenFileDir — Proof Obligation 40

```
assert po40 {
  all sys: System, path: Path, omode: FS_OpenMode {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path]
    => not (isCreateNew[omode]    and isElemFileStore[path, sys.fileStore]) and
       not (isCreateAlways[omode] and isElemTablePath[path, sys.table])     and
       not (isOpen[omode]         and not isElemFileStore[path, sys.fileStore])
       => (isOpen[omode] or omode in FS_OpenAlways) and
          isElemFileStore[path, sys.fileStore]
          => path in RelCalc/dom[sys.fileStore.map]
  }
}
CheckPO40: check po40 for 7
```

Code 4.188: Alloy model sliced at FS_OpenFileDir — Proof Obligation 40

All the POs were model checked with success, that is to say that no counter-examples were found.

4.2.7 VDM++ Adapted for the VdmHolTranslator Tool

This operation introduces more VDM++ constructs that the VdmHolTranslator does not support, so adequate adaptations must be found. However, there are some limitations that can not be overcome through adaptation and some constructs will have to be removed and hand written in HOL syntax later.

Optional data type is the mathematical parametric type

$$[X] \cong 1 + X \tag{4.3}$$

where in VDM the value of type 1 is the literal `nil`. This data type can be rewritten using the sequence type,

$$Y^* \cong 1 + Y \times Y^* \tag{4.4}$$

where the empty sequence represents the `nil` literal, and the only element of a singleton sequence represents the actual value.

Data type `nat1` when used in type declarations, can be adapted using the same strategy as for the `seq1` data type. The type gets rewritten with the `nat` type and an invariant assuring the value to be greater than zero. Functions where the `nat1` type is used for parameters or results, can be rewritten using pre- and post-conditions, respectively, that assure the desired property of the values.

Tuple pattern is also not supported by the translator tool, and of all unsupported constructs this is the only that was not adapted. The tuple patterns, in this specification, are used to pattern match the result of functions that return several values. A possibly equivalent way to rewrite tuples would be to represent them as sequences of values, although this would make the specification prone to errors related with the size of the sequences, that would not be detected by the type checker. It would also lead to the introduction of post-conditions, and stronger pre-conditions. Another option would be to eradicate the functions that return tuples by splitting them in more functions that separately return each element of the tuple. Splitting every function that returns a tuple would lead to an explosion of functions. Due to the presented difficulties the functions that use tuple patterns to match values obtained from another function are removed from the VDM++ model prior to the translation, and hand written in HOL afterwards.

Binary operator `#` that selects an element from a tuple given its position, could be a way to avoid tuple patterns, although it is also not supported. It would be very easy to add support for this operator to the VdmHolTranslator, however there is no equivalent operator in HOL. The translator follows a clever strategy to produce in HOL functions that do field selection on record values, and a similar approach for tuples might also work although it would involve tampering with the translator specification structure and the tool's HOL AST specification.

Set enumeration pattern is used in the `max` function to match a `nat` value inside a set. Using the `let ...be st` construction it is possible to instantiate a value in the give set, hence overcoming the pattern matching limitation.

Symbolic literal pattern is used in the `cases` expression of the `getOpenOffset` function, to match `FS_OpenMode` values. The `cases` expression can be rewritten using simple conditional logic with and `if ... then ... else ...` expression.

After translating the VDM++ OmlAST to the defined HOL AST[4], here on referenced as HolAst, the VdmHolTranslator checks the produced AST to detect missing dependencies. In the case of the VDM++ specification that we want to translate there is a piece of the specification that is not being translated to HolAst, thus producing a dependency error. This is the case of the `fs_open2access_mode_map`, which is a VDM++ value declaration, that the translator tool does not support. The `fs_open2access_mode_map` can be rewritten using a function to apply the same mapping to its parameter. Neither the tool, nor HOL, support `let ... in` statements where variables declared within the statement are used in the declaration of other variables declared within the same statement. To deal with this dependency problem the original `let ... in` statements can be split into statements declared within each other.

4.2.8 Correcting the Translated HOL4 Model

To enable the translation of the specification from VDM++ to HOL, it was necessary to remove two functions: `FS_OpenFileDir_Main` and `FS_OpenFileDir_System`. These two function use tuple patterns to bind local variables to the output of other functions, and tuple patterns are not supported by the VdmHolTranslator. In order to enable a complete translation of the proof obligations they were generated from the complete VDM++ specification, and prior to the translation the two functions were substituted by two stubs with the same data types although no relevant body. Prior to the translation the stubs must be substituted by the appropriate functions in HOL.

```
Define ‘FS_OpenFileDir_System (sys: System)
                              (full_path: Path)
                              (attr: Attributes)
                              (omode: FS_OpenMode) =
  let fs’ = (FS_OpenFileDir_FileStore sys.fileStore full_path attr) and
      (t’,handle) = (FS_OpenFileDir_Table sys.table full_path omode attr.fileType) in
  let offset = (getOpenOffset (FAPPLY fs’ full_path) omode) in
  let t’’ = tableOverride t’ (if   (~(handle = []) /\ (isElemTableHandle (HD handle) t’)
      )
                              then (FEMPTY |+ ((HD handle), ((FAPPLY t’ (HD handle))
                                  with <|fileOffset := (HD offset)|>)))
                              else FEMPTY) in
  ((sys with <|table := t’; fileStore := fs’|>), (HD handle))‘;
```

Code 4.189: FS_OpenFileDir_System hand written in HOL

```
Define ‘FS_OpenFileDir_Main (sys: System)
                            (full_path: Path)
                            (attr: Attributes)
                            (omode: FS_OpenMode) =
  if   ((pre_FS_OpenFileDir_System sys full_path attr omode) /\
        (checkOpenMode sys full_path omode))
  then (if   (mustDeleteFirst sys.fileStore full_path omode) /\
             ((full_path = RootQuoteLiteral) ==> (attr.fileType = DirectoryQuoteLiteral)
                 )
        then (let (sys’,status) = FS_DeleteFileDir_Main sys full_path in
               if   ~(status = FFS_StatusSuccessQuoteLiteral)
                then (sys’, [], status)
```

[4]Within the APS project its author defined an AST of HOL models using VDM++.

```
              else (let (sys'', handle) = FS_OpenFileDir_System sys' full_path attr
                  omode in
                    (sys'', [handle], FFS_StatusSuccessQuoteLiteral)))
        else (let (sys''', handle') = FS_OpenFileDir_System sys full_path attr omode in
              (sys''', [handle'], FFS_StatusSuccessQuoteLiteral)))
  else (let status'' = (FS_OpenFileDir_Exception sys full_path omode) in
        (sys, [], status''))`;
```

Code 4.190: FS_OpenFileDir_Main hand written in HOL

Some of the POs also had to be hand written in the HOL model, because they use unsupported syntactic constructions. Other corrections were performed although other similar corrections have already been discussed in the `FS_DeleteFileDir` section.

4.2.9 Discharging VDM Proof Obligations with HOL4

Just such as it happened with the specification for the `FS_DeleteFileDir` operation, it was necessary to adapt the VDM++ model syntax to comply with the subset of the language that the VdmHolTranslator supports. For this reason some POs are slightly different, and some are not generated for the adapted specification.

Proof Obligation 2: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 10: mapped to a PO in the adapted model, although was not discharged with HOL.

Proof Obligation 11: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 12: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 13: mapped to a PO in the adapted model, and discharged with HOL.

Proof Obligation 20: mapped to a PO in the adapted model, although was not discharged with HOL.

Proof Obligation 22: mapped to a PO in the adapted model, and was discharged with HOL.

Proof Obligation 23: is not generated for the VDM++ specification, and had to be hand written in he HOL model. HOL did not discharge it automatically.

Proof Obligation 24: mapped to a PO in the adapted model, and was discharged with HOL.

Proof Obligation 28: is not generated for the VDM++ specification, and had to be hand written in he HOL model. It was discharged in HOL.

Proof Obligation 30: is not generated for the VDM++ specification, and had to be hand written in he HOL model. HOL did not discharge it automatically.

Proof Obligation 33: mapped to a PO in the adapted model, although was not discharged with HOL.

Proof Obligation 34: mapped to a PO in the adapted model, although was not discharged with HOL.

Proof Obligation 36: mapped to a PO in the adapted model, and was discharged with HOL.

Proof Obligation 40: mapped to a PO in the adapted model, and was discharged with HOL.

From a total of 38 POs, that were either generated from the adapted VDM++ model or hand written in HOL, 24 of them were automatically discharged with HOL, for a full reference of POs and HOL source code see [53].

4.3 Summary

This chapter proposes a VDM++ formal model of an abstract file system following the FSL API from the IFFSCRG document. The model captures the main data structures, and two operations to create and remove files. Two translations are produced in order to apply the verification tool chain, proposed in Chapter 3, to the file system model: manually to Alloy enabling model checking; and automatically to HOL enabling mathematica proof of correction.

Integrating Alloy and VDM++ raises some questions about the level of abstraction used in each specification, because in the perspective of model checking a VDM++ model with and equivalent Alloy model it would make sense to have both models at the same abstraction level. However, Alloy is a declarative language and VDM++ is object-oriented with a functional subset, and this means that usually Alloy models can be more abstract than VDM++ models and still capture the same key aspects. This explains why the option was to have an Alloy model at a higher level of abstraction, when comparing with the "original" VDM++ model.

Integrating HOL and VDM++ benefits from the APS that can automatically translate models from VDM++ to HOL, and produce the adequate proof commands to discharge POs. However, there is still the need to prepare a VDM++ model to be used by the APS translation. Tools to automate the necessary preparation are discussed and presented in the present chapter.

The presented specifications (VDM++, Alloy, and HOL) are intended to be both a contribution to the VFS "mini-challenge" within the GC initiative, and a case study for the verification tool chain presented in the former chapter.

Chapter 5

Related Work and Conclusions

5.1 Related Work

Despite what would be expected there is little work carried out so far on the Verified File System "mini-challenge". At least when compared to previous challenges such as the Mondex. There is a group at the University of York, led by Professor Jim Woodcock, that has not only given a great contribution to the "mini-challenge" by providing Z specifications of file systems at different levels, but also has organized regular workshops and conferences on the subject. In fact it is due to their efforts in studying the problem that made the author of this thesis aware of the IFFSCRG document. The present thesis also relates to other work which, although not focused on the "mini-challenge", uses the Alloy Analyzer as a model checker of specification that are originally written in other formal languages, such as Z or Event B. Finally there can also be established a connection between this thesis and projects that target tool integration in software specification and development.

5.1.1 Verified File System

Woodcock's group at York has been responsible for pushing the "mini-challenge" forward by giving many valuable contributions. They have compiled an extended set of documentation made available both through scientific papers [34, 96] and the web [97]. Moreover, their contribution goes beyond organizing conferences and compiling documentation, as they are currently working on the formalization of file systems at all levels, from the POSIX API down to flash memory devices, following the architecture proposed in [24].

Dr. Andrew Butterfield is both working with the York group and with his students at Dublin in the formalization of the ONFI [99] specification [16]. Work has been done on the formalization of the memory architecture of a flash device, and in the mechanization of proofs in the Z/Eves theorem prover.

The Alloy community at Massachusetts Institute of Technology (MIT), led by Professor Daniel Jackson, has contributed with an Alloy model of a flash file system's basic operations, taking into account the target hardware [48]. They have followed the ONFI [99] specification to model features related with behavior and usability of NAND memories, such as wear leveling and power loss re-

covery. Taking advantage of the Alloy Analyzer the specification has been automatically verified through model checking.

At Southampton, Professor Michael Butler is using the Event-B formal language to tackle the verified file system "mini-challenge". His approach is based on small steps refinement from an abstract tree specification to a file store, attempting to discharge all refinement proof obligations [26].

5.1.2 Using Alloy as a Complement for Other Methods

Alloy is being used by many communities as a complement to their own tools, as it is a very straight forward formal method that combines a powerful declarative language and a fully automated verification tool. This combination is probably what makes Alloy so attractive to users of other formal methods. In the Z community Alloy has been used to verify refinement steps [9] of a simple specification, and is now being used in an industrial-scale case study (see Discussion in [9]). The authors of this work have shed some light on a new application of Alloy to find positive results, mainly because as any other model checker Alloy Analyzer is committed to find negative results such as counter-examples.

The B method community has also considered Alloy as a good choice for model checking their B and Event-B specifications. Work has ben done in the past on Alloy model checking of B specifications [58], and more recently of Event-B [55]. In the latter case, the authors propose an encoding of a Event-B specification in Alloy, so that some invariants that the RODIN tool [1] could not automatically discharge would be validated through model checking in the Alloy Analyzer. The encoding they propose is in many cases similar to what has been proposed in this thesis regarding VDM++ to Alloy translation. They have plans for incorporating a translator from Event-B to Alloy in the RODIN tool set. The same kind of work has been done for the informal and graphical UML notation [79]. However, the results reported by the authors of such study have not been so positive as for other notations.

5.2 Conclusions

5.2.1 Contributions

The specifications built in the scope of this project model a simplistic and abstract file system supporting creation and removal of file system objects according to the requirements found in IFF-SCRG. Following a strategy known as *objectification* [25], an object-oriented API specification in VDM++ was built on top of a purely functional behavior specification. This structure allowed the implementation of a state-based API that is amenable to functional verification by abstracting the notion of state. Soundness properties were captured by data type invariants assuring overall consistency of the model, and behavior by abstract relations in Alloy and explicit functions in VDM++, equipped with adequate pre- and post-conditions in both cases.

The verification process presented in this thesis was devised to validate the abstract file system

specification as a proof of concept. The specification was first validated through unit testing using the VDMUnit framework that tests all functions that are involved in the specification of the objectified operations, as shown in Tables 4.4 and 4.8.

The specification continued to be verified by model checking through a translation of the VDM++ model to an Alloy model, which could be automatically verified in the Alloy Analyzer. The resulting Alloy model is more abstract and in some cases purely relational, in the sense that VDM++ functions are specified as relations among state, and both input and output parameters. Having an Alloy model completely written in purely relational point free notation would be useful to integrate with theorem proving tools that are being built around such mathematical abstractions [63]. Taking advantage of the model finding features of the Alloy Analyzer, the specification was checked for functional requirements and satisfiability. All proof obligations generated within the VDMTools from the VDM++ model were encoded in Alloy, and model checked.

Furthermore, 24 out of 38 proof obligations were mechanically discharged in HOL with the Overture APS. In order to increase the tool's productivity two parsers were built: one to extract the POs from a file generated by the VDMTools; and another to extract the PO expressions from an OmlAst representation. The VDMTools, the Overture Parser and the two new parsers were combined in a script that automatically performs all the preparation steps prior to the translation.

Regarding tool integration, it was shown that the translation of a functional VDM++ model to an Alloy model can be straightforward if considering only data types and predicate skeletons. Although Alloy is a declarative language and VDM++ is an object-oriented language, the construction of the VDM++ model can be done in a way that allows for an easier translation to Alloy. Being that the Overture project has already made the link to the HOL theorem prover available, and that a similar tool could also be built to link VDM++ to Alloy, it seems fair to say that the integration of so different formal methods such as VDM++, Alloy and HOL with the respective tools can be achievable.

The verification tool chain for VDM++ models, presented in Chapter 3, aims at gradually increasing confidence in the consistency of such models, however it could be reorganized differently (see Figure 5.1) with Alloy at the top, followed by VDM and HOL respectively. In this new configuration, requirements would be captured in a purely relational Alloy model, abstracting away from any data structures whatsoever. Requirements would be verified by model checking before, stepping through from the Alloy to VDM++ model, that would be an exercise of refinement, instead of translation. At this point the POs could be generated, and discharged in HOL4 using the APS.

5.2.2 Difficulties

Program slicing is a technique for decomposition or simplification of programs, where the intention is to focus on specific parts of the program. It can be applied forwardly, backwardly and in both ways for the smallest slice of the code, that accommodates a given semantical aspect. This project would have greatly benefited from a slicing tool both for VDM++ and Alloy. Whenever doing verification of elaborate models through the verification of individual properties, it is useful to work with the smallest (sub)models that accommodate each property being checked. This allows for better scalability of verification, by reducing the size of a specification to the minimum necessary, hence

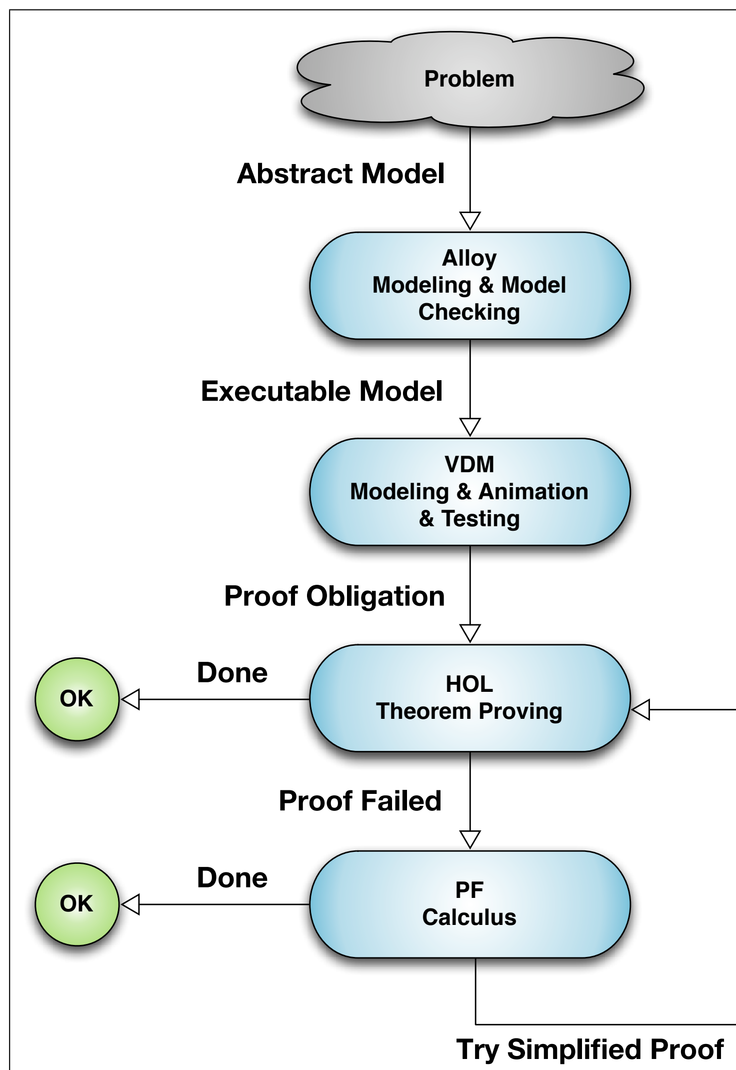

Figure 5.1: Alloy — VDM — HOL Tool Chain

reducing memory and Central Processing Unit (CPU) time needed for verification. Throughout this project, specifications have been sliced manually both for analysis and verification purposes, and this has had a significant impact on their readability and verifiability. On the other hand, manually slicing also had a negative impact to the overall productivity, because it consumed a quite significant amount of time.

Formal specification languages are usually good to express abstract properties of systems, however they are not pure mathematics and therefore each has its implementation details. Formally specifying software in different languages is a good way to gain greater insight into its abstract properties, because the implementation details of each language become easier to distinguish from the kernel of the abstract specification. However, to achieve the integration of different languages it is necessary to have tools for automatic synchronization among them, otherwise as size and complexity increases, manual synchronization becomes unfeasible.

Tool maturity is also a fundamental aspect of software development. In this project, we found

the VDMTools and the Alloy Analyzer mature enough, contrary to the Overture APS which is still unripe. The tools one chooses to achieve one's results are decisive to obtain success, so tools not mature enough to support development beyond stack and queue examples can condemn projects to failure. Much time of this project was invested, after all, in mastering the Overture Automatic Proof System. Thanks to a lot of help from its author it has become a key stone of the proposed verification process. However, the learning curve associated with this tool requires a lot of investment, as it is still necessary to adjust VDM++ models, correct HOL models, manually apply proof tactics to obtain a true negative result from a failed proof. The main difficulty in using the system is related with the preparation of a VDM++ model to be translated to HOL, together with the desired proof obligations. Although this is not a complex task, it is quite laborious and not well documented. Furthermore, it is a process that needs to be repeated as many times as the specifications changes and need to be verified. This thesis contributed to further automate the preparation of VDM++ models for the APS. The integration of these automations within the APS project was one of the goals of the The Fifth Overture Workshop, recently held in Portugal.

Refinement of the top-level file system abstract specification was one of the initial goals of the project which did not come through. Another was the verification in Alloy and HOL of such refinement process, to see if the proposed verification process for satisfiability proof obligations scales up to refinement proof obligations. Although the author has paid attention to the lower layers in order to achieve the desired multi-layered architecture thorough refinement steps, completing the exercise proved to be unfeasible within the project's overall schedule and time-span.

5.2.3 Future Work

As future work on the verification of the flash file system specification, more operations should be modeled and verified. The current VDM++ specification already includes the `FS_WriteFile` and the `FS_SetFileOffset` [53] operations together with the respective unit tests. To complete the verification cycle for these operations it is still necessary to model check them in Alloy, and attempt mechanical proof of obligations in HOL. Regarding the operations already verified there are still proof obligations that, despite having been model checked with success, still haven't been discharged through mathematical proof. So they call for the last ingredient of the overall life-cycle proposed in this thesis and in [30], that of manually proving them using the PF-transform, as already shown in [66].

Once a minimum working subset of top-level API operations is modeled and verified, the whole multi-layered structure of IFFSCRG should be addressed from a data refinement point of view, as already mentioned. In a bottom-up approach, one has to reverse engineer the lower levels and show that they are valid refinements of the upper ones. In a top-down way, this means specifying the complete architecture by refining the abstract file system model into more concrete models of it, one sub layer being added at a time. Only after this (expectedly sizable) piece of work is carried out can one offer Joshi and Holzmann what they have asked for when putting forward the VFS "mini-challenge" [47].

Bibliography

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In Börger et al. [10], page 347.

[2] M. Bauer, R. Alexis, Greg Atwood, B. Baltar, Al Fazio, K. Frary, M. Hensel, M. Ishac, J. Javanifard, M. Landgraf, D. Leak, K. Loe, Duane Mills, P. Ruby, R. Rozman, S. Sweha, S. Talreja, and K. Wojciechowski. A Multilevel-Cell 32MB Flash Memory. In *ISMVL*, pages 367–, 2000.

[3] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In Lionel C. Briand and Alexander L. Wolf, editors, *FOSE*, pages 85–103, 2007.

[4] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[5] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. In *Proceedings of the IEEE*, volume 91, pages 489–502. Central Res. & Dev. Dept., STMicroclectronics, Agrate Brianza, Italy, IEEE, 2003.

[6] Juan Bicarregui. The Verified Software Repository. Website: `http://vsr.sourceforge.net/`, August 2007.

[7] Juan Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Asp. Comput.*, 18(2):143–151, 2006.

[8] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[9] Christie Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electr. Notes Theor. Comput. Sci.*, 137(2):23–44, 2005.

[10] Egon Börger, Michael Butler, Jonathan 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.

[11] Allan G. Bromley. The evolution of Babbage's calculating engines. *IEEE Ann. Hist. Comput.*, 9(2):113–136, 1987.

[12] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[13] Ricky W. Butler. NASA LaRC Formal Methods Program. Website:
`http://shemesh.larc.nasa.gov/fm/`, January 2008.

[14] Ricky W. Butler and George B. Finelli. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Trans. Software Eng.*, 19(1):3–12, 1993.

[15] Ricky W. Butler and Sally C. Johnson. Formal Methods for Life-Critical Software. Technical report, NASA Langley Research Center, 1993.

[16] Andrew Butterfield and Jim Woodcock. Formalising Flash Memory: First Steps. In *ICECCS*, pages 251–260. IEEE Computer Society, 2007.

[17] Michael Holloway C. Why Engineers Should Consider Formal Methods. Technical report, NASA Langley Research Center, 1997.

[18] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[19] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[20] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[21] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[22] UK Computer Research Committee. Grand challenges in computing research - the iet. Website:
`http://www.ukcrc.org.uk/grand_challenges/`.

[23] Intel Corporation. Intel museum. Website:
`http://www.intel.com/museum/`.

[24] Intel Corporation. *Intel Flash File System Core Reference Guide*, October 2004. Doc. Ref. 304436-001.

[25] A. Miguel Cruz, Luís Soares Barbosa, and José Nuno Oliveira. From algebras to objects: Generation and composition. *J. UCS*, 11(10):1580–1612, 2005.

[26] Kriangsak Damchoom, Michael Butler, and Jean-Raymond 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.

[27] Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, Graham Robinson, Michael J. C. Gordon, and Thomas F. Melham. The PROSPER Toolkit. In

Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2000.

[28] Bruno Dias and Miguel Ferreira. NAND Flash Interface Specification. Technical report, University of Minho, 2007.

[29] David L. Dill and John Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. *IEEE Computer*, 29(4):23–24, apr 1996.

[30] Miguel Ferreira, Samuel Silva, and José Nuno Oliveira. Verifying intel flash file system core specification. *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, May 2008.

[31] Kate Finney and Norman E. Fenton. Evaluating the Effectiveness of Z: The Claims Made About CICS and Where We Go From Here. *Journal of Systems and Software*, 35(3):209–216, 1996.

[32] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.

[33] Leo Freitas, Zheng Fu, and Jim Woodcock. POSIX file store in Z/Eves: an experiment in the verified software repository. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Leo Freitas, Jim Woodcock, and Andrew Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 153–162, Washington, DC, USA, 2008. IEEE Computer Society.

[35] Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Softw.*, 25(5):30–37, 2008.

[36] Michael J. C. Gordon. Introduction to the HOL System. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *TPHOLs*, pages 2–3. IEEE Computer Society, 1991.

[37] Mike Gordon. *From LCF to HOL: a short history*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[38] CSK Group. Csk holdings. Website:
`http://www.csk.com`.

[39] CSK Group. Website:
`http://www.vdmtools.jp/`, 2008.

[40] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[41] Tony Hoare and Jay Misra. *Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project*, pages 1–18. Springer-Verlag, Berlin, Heidelberg, 2005.

[42] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[43] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Heyward Street, Cambridge, MA02142, USA, April 2006.

[44] David Janzen and Hossein Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *IEEE Computer*, 38(9):43–50, 2005.

[45] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[46] Cliff B. Jones. Scientific Decisions which Characterize VDM. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 1999.

[47] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Asp. Comput.*, 19(2):269–272, 2007.

[48] Eunsuk Kang and Daniel Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In Börger et al. [10], pages 294–308.

[49] Hyeong-Ju Kang and In-Cheol Park. SAT-based unbounded symbolic model checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(2):129–140, 2005.

[50] Gerwin Klein, Steve Rowe, and Régis Décamps. JFLEX - The Fast Scanner Generator for Java. Website:
`http://jflex.de/`, May 2008.

[51] Software Quality Research Laboratory. Pacemaker Formal Methods Challenge. Website:
`http://www.cas.mcmaster.ca/sqrl/pacemaker.htm`, April 2007.

[52] Stefan K. Lai. Flash memories: Successes and challenges. *IBM Journal of Research and Development*, 52(4/5):529–535, 2008.

[53] Peter Grom Larsen. VDM Portal. Website:
`http://www.vdmportal.org`.

[54] Hugo Daniel Macedo, Peter Gorm Larsen, and John S. Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[55] Paulo J. Matos and João Marques-Silva. Model Checking Event-B by Encoding into Alloy. *CoRR*, abs/0805.3256, 2008.

[56] Kenneth L. McMillan. *Interpolation and SAT-Based Model Checking* , volume 2725/2003 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2004.

[57] Bertrand Meyer. Seven Principles of Software Testing. *Computer*, 41(8):99–101, 2008.

[58] Leonid Mikhailov and Michael J. Butler. An Approach to Combining B and Alloy. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 140–161. Springer, 2002.

[59] Robert Milne. Proof Rules for VDM Statements. In Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones, editors, *VDM Europe*, volume 328 of *Lecture Notes in Computer Science*, pages 318–336. Springer, 1988.

[60] Carroll Morgan and Bernard Sufrin. Specification of the UNIX Filing System. *IEEE Trans. Software Eng.*, 10(2):128–142, 1984.

[61] Paul Mukherjee, Fabien Bousquet, Jerome Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[62] Madan Musuvathi. Systematic concurrency testing using CHESS. In Shmuel Ur, editor, *PADTAD*, page 10. ACM, 2008.

[63] C. Necco, J.N. Oliveira, and J. Visser. ESC/PF: Static checking of relational models by calculation, 2008. (Submitted).

[64] Thomas R. Nicely. Thomas R. Nicely's Home Page. Website: `http://www.trnicely.net/`, August 2008.

[65] Michael Norrish and Konrad Slind. Hol 4 kananaskis 4. Website: `http://hol.sourceforge.net`.

[66] J.N. Oliveira. Extended static checking by calculation using the pointfree transform, 2008. Tutorial paper (56 p.) accepted for publication by Springer-Verlag, LNCS series.

[67] Aleph One. Yaffs - a flash file sysem for embedded use. Website: `http://www.yaffs.net/`.

[68] Overture-Core-Team. Overture web site. Website: `http://www.overturetool.org`, 2007.

[69] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[70] Patrick R.H. Place. POSIX 1003.21—Real Time Distributed Systems Communication. Technical report, Software Engineering Institute, Carnegie Mellon University, August 1995.

[71] Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Notices*, 27(8):76–82, 1992.

[72] A Rahman, R Haque, and K Tedrow. Memory array with pseudo single bit memory cell and method. *US Patent 7272041*, September 2007.

[73] Glenn E. Reeves and Tracy A. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199, 2005.

[74] Neal R. Reizer, Gregory D. Abowd, B. Craig Meyers, and Patrick R.H. Place. Using formal methods for requirements specification of a proposed POSIX standard. *Proceedings of the First International Conference on Requirements Engineering*, pages 118–125, 1994.

[75] Sergei Romanenko, Claudio Russo, and Peter Sestoft. Moscow ML Language Overview. Website:
`http://www.itu.dk/~sestoft/mosml.html`, June 2000.

[76] Mark Saaltink. The Z/EVES System. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

[77] Adriana Sucena Santos. VDM++ Test Automation Support. Master's thesis, Minho University with exchange to Engineering College of Arhus, July 2008.

[78] Dana S. Scott. A Type-Theoretical Alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1&2):411–440, 1993.

[79] Anthony J. H. Simons and Carlos Alberto Fernandez y Fernandez. Using Alloy to model-check visual design notations. *CoRR*, abs/0802.2258, 2008.

[80] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[81] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[82] Open Group Technical Standard. Standard for information technology - Portable operating system interface (POSIX). Base Definitions. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. System Interfaces*, 2004.

[83] Open Group Technical Standard. Standard for information technology - Portable operating system interface (POSIX). Rationale (Informative). *IEEE Std 1003.1, 2004 Edition. The Open*

Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. System Interfaces, 2004.

[84] Open Group Technical Standard. Standard for information technology - Portable operating system interface (POSIX). System Interfaces. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. System Interfaces*, 2004.

[85] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In Paolo Camurati and Hans Eveking, editors, *CHARME*, volume 987 of *Lecture Notes in Computer Science*, pages 21–34. Springer, 1995.

[86] Mana Taghdiri and Daniel Jackson. A Lightweight Formal Analysis of a Multicast Key Management Scheme. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *FORTE*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2003.

[87] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, January 1936.

[88] Manuel van den Berg, Marcel Verhoef, and Mark Wigmans. Formal Specification and Development of a Mission Critical Data Handling Subsystem – an Industrial Usage Report. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 95–98, September 1999.

[89] Manuel van den Berg, Marcel Verhoef, and Mark Wigmans. Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 85–93, September 1999.

[90] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[91] Sander Vermolen. Automated Proof Support. Website:
`http://www.overturetool.org/twiki/bin/view/Main/AutomaticProof`.

[92] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007.

[93] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, 1993.

[94] Maurice V. Wilkes. Babbage's Expectations for his Engines. *IEEE Ann. Hist. Comput.*, 13(2):141–145, 1991.

[95] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2000.

[96] Jim Woodcock. First Steps in the Verified Software Grand Challenge. *IEEE Computer*, 39(10):57–64, 2006.

[97] Jim Woodcock and Leonardo Freitas. ABZ Conference 2008 - ASM + B + Z. Website: `http://www.cs.york.ac.uk/circus/mc/abz`, 2008.

[98] David Woodhouse. JFFS: The Journalling Flash File System. Technical report, Red Hat, Inc, 2001.

[99] ONFI Workgroup. Open NAND Flash Interface Specification. Technical Report 2.0, Hynix Semiconductor and Intel Corporation and Micron Technology, Inc. and Phison Electronics Corp. and Sony Corporation and Spansion and STMicroelectronics, February 2008.

[100] Pamela Zave. A Formal Model of Addressing for Interoperating Networks. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2005.

Appendix A

Libraries

A.1 Alloy Relational Calculus Library

```
module RelCalc
/*
 * Authors:
 *      Miguel Ferreira         <miguel@di.uminho.pt>
 *      Samuel Silva            <silva.samuel@gmail.com>
 * Description:
 *          Relational calculus library.
 * Conventions:
 *      - all relations begin with a capital letter;
 *      - all functions begin with a regular letter.
 */

/**********************************************/

fun id[S: univ] : univ -> univ {
    (S -> S) & iden
}

fun ker [R : univ -> univ] : univ -> univ {
    R . (~R)
}
fun img [R : univ -> univ] : univ -> univ {
    (~R) . R
}

fun dom [R: univ->univ] : set (R.univ) {
    R.univ
}

fun rng [R: univ->univ] : set (univ.R) {
    univ.R
}

/**********************************************/

pred Reflexive [R : univ -> univ, S: set univ] { id[S] in R }

pred Correflexive [R : univ -> univ, S: set univ] { R in id[S] }
```

```
pred Symmetric [R : univ -> univ] { R in ~R }

pred Transitive [R : univ -> univ] { R.R in R }

pred Cotransitive [R: univ -> univ] { R in R.R }

pred Antisymmetric [R: univ -> univ, S: univ] { R & ~R in id[S] }

pred Per [R: univ -> univ] {
    Symmetric[R]
    Transitive[R]
    Cotransitive[R]
}

pred Preorder [R: univ -> univ, S: set univ] {
    Transitive[R]
    Reflexive[R,S]
}
pred Equivalence [R: univ -> univ, S: set univ] {
    Per[R]
    Preorder[R,S]
}

pred Partialorder [R: univ -> univ, S: set univ] {
    Preorder[R,S]
    Antisymmetric[R,S]
}

pred Id [R: univ -> univ, S: set univ] {
    Correflexive[R,S]
    Equivalence[R,S]
    Partialorder[R,S]
}

pred Linearorder [R: univ -> univ, S: set univ] {
    Partialorder[R,S]
    Connected[R,S]
}

pred Connected [R : univ -> univ, S: set univ] {
    (R + ~R) = (S -> S)
}

pred Simple [R: univ -> univ, S: set univ] { Correflexive[img[R],S] }

pred Entire [R: univ -> univ, S: set univ] { Reflexive[ker[R],S] }

pred Surjective [R: univ -> univ, S: set univ] { Reflexive[img[R],S] }

pred Injective [R: univ -> univ, S: set univ] { Correflexive[ker[R],S] }

pred Function [R: univ -> univ, A,B: set univ] {
    Simple[R,B]
    Entire[R,A]
}

pred Representation [R: univ -> univ, A: set univ] {
```

```
    Injective[R,A]
    Entire[R,A]
}

pred Abstraction [R: univ -> univ, B: set univ] {
    Simple[R,B]
    Surjective[R,B]
}

pred Bijection [R: univ -> univ, A, B: set univ] {
    Representation[R,A]
    Abstraction[R,B]
}

/***********************************************/

pred Assymetric [R: univ -> univ] { ~R not in R }

pred Acyclic [R: univ -> univ, S: set univ] { no ^R & id[S] }

/***********************************************/

fun converse [R: univ -> univ] : univ -> univ { ~R }

/***********************************************/

run {}
```

Appendix B

Models

B.1 VDM++ Model

B.1.1 Flash File System Core

```
class FlashFileSystemCore


types

protected
FFS_Status =
    <FFS_StatusSuccess>
  | <FS_ErrorFileNotFound>
  | <FS_ErrorFileStillOpen>
  | <FS_ErrorDirectoryNonEmpty>
  | <FS_ErrorFileAlreadyExists>
  | <FS_ErrorInvalidPath>
  | <FFS_StatusInvalidParameter>
  | <FFS_StatusUnknown>;

end FlashFileSystemCore
```

File System Layer Base

```
class FileSystemLayerBase is subclass of FlashFileSystemCore


types

protected
System :: table     : OpenFilesTable
          fileStore : FileStore
inv sys ==
  forall ofi in set rng sys.table &
    isElemFileStore(ofi.path, sys.fileStore) and
    isRegularFile(sys.fileStore(ofi.path).info);

protected
```

```
FileStore = map Path to File
inv fileStore ==
  forall path in set dom fileStore &
    let parent = dirName(path) in
    parent in set dom fileStore         and
    isDirectory(fileStore(parent).info);

protected
Path = <Root> | seq1 of FileName;

protected
FileName = seq1 of char;

protected
File ::
  info     : FS_FileDirInfo
  contents : [FileContents]
inv file ==
  (isDirectory(file.info)   and file.contents =  nil) or
  (isRegularFile(file.info) and file.contents <> nil);

protected
FS_FileDirInfo :: attributes : Attributes;

protected
Attributes :: fileType : FileType;

protected
FileType = <RegularFile> | <Directory>;

protected
OpenFilesTable = map FS_FileHandle to FS_OpenFileInfo;

protected
FS_FileHandle = nat;

protected
FS_OpenFileInfo ::
  fileOffset : nat1
  accessMode : FS_AccessMode
  path       : Path;

protected
FS_OpenMode =
    <FS_CreateNew>
  | <FS_CreateAlways>
  | <FS_OpenRead>
  | <FS_OpenWrite>
  | <FS_OpenAlways>
  | <FS_OpenWriteOnly>
  | <FS_CreateAlwaysReadOnly>
  | <FS_CreateNewReadOnly>;

protected
FS_AccessMode =
    <FS_AccessReadOnly>
  | <FS_AccessWriteOnly>
  | <FS_AccessReadWrite>;
```

```
protected
FileContents = seq of token;


functions

protected
dirName : Path -> Path
dirName(full_path) ==
  cases full_path:
    <Root> -> <Root>,
    [-]    -> <Root>,
    others -> [ full_path(i) | i in set inds full_path & i < len full_path ]
  end;

protected
isDirectory : FS_FileDirInfo -> bool
isDirectory(info) == info.attributes.fileType = <Directory>;

protected
isRegularFile : FS_FileDirInfo -> bool
isRegularFile(info) == info.attributes.fileType = <RegularFile>;

protected
hasSubFiles : FileStore * Path -> bool
hasSubFiles(fileStore, path) ==
  exists subpath in set dom fileStore & path = dirName(subpath);

protected
isRoot : Path -> bool
isRoot(path) == path = <Root>;

protected
isElemFileStore : Path * FileStore -> bool
isElemFileStore(path, fileStore) == path in set dom fileStore;

protected
isElemTablePath: Path * OpenFilesTable -> bool
isElemTablePath(path, table) == exists ofd in set rng table & ofd.path = path;

protected
isElemTableHandle: FS_FileHandle * OpenFilesTable -> bool
isElemTableHandle(handle, table) == handle in set dom table;

protected
isDirName : Path * Path -> bool
isDirName(parent, path) == parent = dirName(path);

protected
newFileDirInfo: Attributes -> FS_FileDirInfo
newFileDirInfo(attributes) ==
  mk_FS_FileDirInfo(attributes);

protected
emptyFileContents : FileType -> [FileContents]
emptyFileContents(fileType) ==
  if fileType = <Directory> then nil else "";
```

```
protected
newFileHandle : set of FS_FileHandle -> FS_FileHandle
newFileHandle(handles) ==
  if card handles = 0 then 1 else max(handles) + 1
post RESULT not in set handles;

protected
max : set of nat -> nat
max(s) ==
  let {result} = { x | x in set s & forall y in set s & x >= y } in
  result
pre card s > 0;


values

protected
fs_open2access_mode_map: map FS_OpenMode to FS_AccessMode =
{
  <FS_CreateNew>             |-> <FS_AccessReadWrite>,
  <FS_CreateAlways>          |-> <FS_AccessReadWrite>,
  <FS_OpenRead>              |-> <FS_AccessReadOnly>,
  <FS_OpenWrite>             |-> <FS_AccessReadWrite>,
  <FS_OpenAlways>            |-> <FS_AccessReadWrite>,
  <FS_OpenWriteOnly>         |-> <FS_AccessWriteOnly>,
  <FS_CreateAlwaysReadOnly>  |-> <FS_AccessReadOnly>,
  <FS_CreateNewReadOnly>     |-> <FS_AccessReadOnly>
};

protected
initFileStore = {<Root> |-> mk_File(newFileDirInfo(mk_Attributes(<Directory>)), nil)};

protected
initOpenFilesTable = {|->};

protected
fileAttributes = mk_Attributes(<RegularFile>);

protected
dirAttributes = mk_Attributes(<Directory>);

end FileSystemLayerBase
```

File System Layer Operations

```
class FileSystemLayerOperations is subclass of FileSystemLayerBase


functions

protected
FS_DeleteFileDir_Main: System * Path -> System * FFS_Status
FS_DeleteFileDir_Main(sys, full_path) ==
  if full_path in set dom sys.fileStore and
     pre_FS_DeleteFileDir_System(sys, full_path)
```

```
  then mk_(FS_DeleteFileDir_System(sys, full_path), <FFS_StatusSuccess>)
  else mk_(sys, FS_DeleteFileDir_Exception(sys, full_path));

protected
FS_DeleteFileDir_System: System * Path -> System
FS_DeleteFileDir_System(sys, full_path) ==
  mu(sys, fileStore |-> FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}))
pre (forall ofi in set rng sys.table & ofi.path <> full_path) and
    pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path});

protected
FS_DeleteFileDir_FileStore: FileStore * set of Path -> FileStore
FS_DeleteFileDir_FileStore(fileStore, paths) ==
  paths <-: fileStore
pre forall path in set dom fileStore &
    dirName(path) in set paths => path in set paths;

protected
FS_DeleteFileDir_Exception: System * Path -> FFS_Status
FS_DeleteFileDir_Exception(sys, full_path) ==
  if not isElemFileStore(full_path, sys.fileStore)
  then <FS_ErrorFileNotFound>
  elseif isElemTablePath(full_path, sys.table)
  then <FS_ErrorFileStillOpen>
  elseif isDirectory(sys.fileStore(full_path).info) and
         hasSubFiles(sys.fileStore, full_path)
  then <FS_ErrorDirectoryNonEmpty>
  else <FFS_StatusUnknown>;

protected
FS_OpenFileDir_Main: System * Path * Attributes * FS_OpenMode
  -> System * [FS_FileHandle] * FFS_Status
FS_OpenFileDir_Main(sys, full_path, attributes, omode) ==
  if   pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
       checkOpenMode(sys, full_path, omode)
  then if   mustDeleteFirst(sys.fileStore, full_path, omode) and
            (full_path = <Root> => attributes.fileType = <Directory>)
       then let mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path) in
            if   status <> <FFS_StatusSuccess>
            then mk_(sys', nil, status)
            else let result = FS_OpenFileDir_System(sys', full_path, attributes, omode)
                in
                       mk_(result.#1, result.#2, <FFS_StatusSuccess>)
       else let mk_(sys'',handle) = FS_OpenFileDir_System(sys, full_path, attributes,
           omode) in
            mk_(sys'', handle, <FFS_StatusSuccess>)
  else mk_(sys, nil, FS_OpenFileDir_Exception(sys, full_path, omode));

protected
mustDeleteFirst: FileStore * Path * FS_OpenMode -> bool
mustDeleteFirst(fs, path, omode) ==
  isCreateAlways(omode) and isElemFileStore(path, fs);

protected
checkOpenMode: System * Path * FS_OpenMode -> bool
checkOpenMode(sys, path, omode) ==
  not (isCreateNew(omode)    and isElemFileStore(path, sys.fileStore))     and
  not (isCreateAlways(omode) and isElemTablePath(path, sys.table))         and
```

```
  not (isOpen(omode)          and not isElemFileStore(path, sys.fileStore)) and
  not ((isOpen(omode) or omode = <FS_OpenAlways>) and
       isElemFileStore(path, sys.fileStore)       and
       not isRegularFile(sys.fileStore(path).info));

protected
FS_OpenFileDir_System: System * Path * Attributes * FS_OpenMode
  -> System * [FS_FileHandle]
FS_OpenFileDir_System(sys, full_path, attr, omode) ==
  let fileStore'        = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset            = getOpenOffset(fileStore'(full_path), omode),
      table'            = table ++ if   handle <> nil and isElemTableHandle(handle,
          table)
                                   then {handle |-> mu(table(handle), fileOffset |->
                                       offset)}
                                   else {|->} in
  mk_(mk_System(table', fileStore'), handle)
pre pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr);

protected
getOpenOffset: File * FS_OpenMode -> [nat1]
getOpenOffset(file, omode) ==
  if   file.contents <> nil
  then cases omode:
         <FS_OpenWrite>  -> (len file.contents) + 1,
         <FS_OpenAlways> -> (len file.contents) + 1,
         others          -> 1
       end
  else nil;

protected
FS_OpenFileDir_FileStore: FileStore * Path * Attributes -> FileStore
FS_OpenFileDir_FileStore(fileStore, full_path, attributes) ==
  if   not isElemFileStore(full_path, fileStore)
  then let content = emptyFileContents(attributes.fileType),
           newFile = mk_File(newFileDirInfo(attributes), content) in
       fileStore munion { full_path |-> newFile }
  else fileStore
pre (full_path = <Root> and attributes.fileType = <Directory>) or
    (let parent = dirName(full_path) in
     isElemFileStore(parent,fileStore) and isDirectory(fileStore(parent).info));

protected
FS_OpenFileDir_Table: OpenFilesTable * Path * FS_OpenMode * FileType -> OpenFilesTable *
     [FS_FileHandle]
FS_OpenFileDir_Table(table, full_path, omode, fileType) ==
  if   fileType = <Directory>
  then mk_(table, nil)
  else let amode  = fs_open2access_mode_map(omode),
           ofi    = mk_FS_OpenFileInfo(1, amode, full_path),
           handle = newFileHandle(dom table) in
       mk_(table munion { handle |-> ofi }, handle);

protected
FS_OpenFileDir_Exception: System * Path * FS_OpenMode -> FFS_Status
FS_OpenFileDir_Exception(sys, full_path, omode) ==
```

```
  if      isCreateNew(omode) and isElemFileStore(full_path, sys.fileStore)
  then    <FS_ErrorFileAlreadyExists>
  elseif isCreateAlways(omode) and isElemTablePath(full_path, sys.table)
  then    <FS_ErrorFileStillOpen>
  elseif isOpen(omode) and not isElemFileStore(full_path, sys.fileStore)
  then    <FS_ErrorFileNotFound>
  elseif isOpen(omode) and isElemFileStore(full_path, sys.fileStore) and
         not isRegularFile(sys.fileStore(full_path).info)
  then    <FFS_StatusInvalidParameter>
  elseif full_path <> <Root> and not isElemFileStore(dirName(full_path), sys.fileStore)
  then    <FS_ErrorInvalidPath>
  elseif full_path <> <Root> and not isDirectory(sys.fileStore(dirName(full_path)).info)
  then    <FS_ErrorInvalidPath>
  else    <FFS_StatusUnknown>;

protected
isCreateNew: FS_OpenMode -> bool
isCreateNew(omode) ==
  omode = <FS_CreateNew> or
  omode = <FS_CreateNewReadOnly>;

protected
isCreateAlways: FS_OpenMode -> bool
isCreateAlways(omode) ==
  omode = <FS_CreateAlways> or
  omode = <FS_CreateAlwaysReadOnly>;

protected
isOpen: FS_OpenMode -> bool
isOpen(omode) ==
  omode = <FS_OpenRead>  or
  omode = <FS_OpenWrite> or
  omode = <FS_OpenWriteOnly>;

protected
FS_Init_Main : () -> System * FFS_Status
FS_Init_Main() == mk_(FS_Init_System(), <FFS_StatusSuccess>);

protected
FS_Init_System : () -> System
FS_Init_System() == mk_System(FS_Init_Table(), FS_Init_FileStore());

protected
FS_Init_Table : () -> OpenFilesTable
FS_Init_Table() == {|->};

protected
FS_Init_FileStore : () -> FileStore
FS_Init_FileStore() ==
  {<Root> |-> mk_File(mk_FS_FileDirInfo(mk_Attributes(<Directory>)), nil)};

end FileSystemLayerOperations
```

File System Layer Object

```
class FileSystemLayerObject is subclass of FileSystemLayerOperations
```

```
instance variables

protected
sys : FileSystemLayerBase`System := FS_Init_Main().#1;


operations

public
SetSystem: System ==> ()
SetSystem(sys') ==
  sys := sys';


public
FS_DeleteFileDir : Path ==> FFS_Status
FS_DeleteFileDir(full_path) ==
  def mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path) in
  (sys := sys'; return status);

public
FS_OpenFileDir : Path * Attributes * FS_OpenMode
  ==> [FS_FileHandle] * FFS_Status
FS_OpenFileDir(full_path, attributes, omode) ==
  def mk_(sys',handle,status)
        = FS_OpenFileDir_Main(sys, full_path, attributes, omode) in
  (sys := sys'; return mk_(handle, status));

end FileSystemLayerObject
```

B.2 Unit tests

B.2.1 FS_DeleteFileDir

Intel Test

```
class IntelTest

operations

public
Execute: () ==> ()
Execute() ==
  (dcl ts : TestSuite := new TestSuite();
       ts.AddTest(new FileStoreTest("FileStore Test"));
       ts.AddTest(new SystemTest("System Test"));
       ts.AddTest(new DeleteFileDirTest("FS_DeleteFileDir Test"));

       ts.Run())

end IntelTest
```

System Test

```
class SystemTest is subclass of TestCase, FileSystemLayerOperations


instance variables

protected
vals : FileSystemLayerValues;

protected
sys : FileSystemLayerBase`System;

protected
io : IO;


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   sys  := mk_System({|->},vals.fs1);
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (SimpleDeleteTest(["etc","hosts"]);
   SimpleDeleteTest(["etc","conf.d"]);
   SimpleDeleteTest(["etc","resolv.conf"]);
   SimpleDeleteTest(["bin","ls"]);
```

```
   SimpleDeleteTest(["bin","wc"]);
   SimpleDeleteTest(["etc"]);
   SimpleDeleteTest(["bin"]);
   SimpleDeleteTest(<Root>));

protected
TearDown: () ==> ()
TearDown() == skip;

protected
SimpleDeleteTest: Path ==> ()
SimpleDeleteTest(path) ==
  let sys' = FS_DeleteFileDir_System(sys,path) in
  (AssertTrue(path not in set dom sys'.fileStore);
   AssertTrue(dom sys.fileStore = dom sys'.fileStore union {path});
   AssertTrue(sys.table = sys'.table);
   sys := sys');

end SystemTest
```

File Store Test

```
class FileStoreTest is subclass of TestCase, FileSystemLayerOperations


instance variables

protected
vals : FileSystemLayerValues;

protected
fs : FileStore;

protected
io : IO;


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   fs   := vals.fs1;
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (dcl init : FileStore := fs;
   SimpleDeleteTest({["etc", "hosts"]});
   SimpleDeleteTest({["etc", "conf.d"]});
   SimpleDeleteTest({["bin", "ls"]});
   SimpleDeleteTest({<Root>, ["etc"], ["etc", "resolv.conf"], ["bin"], ["bin","wc"]});
  );
```

```
protected
TearDown: () ==> ()
TearDown() == skip;

protected
SimpleDeleteTest: set of Path ==> ()
SimpleDeleteTest(paths) ==
  let fs' = FS_DeleteFileDir_FileStore(fs,paths) in
  (AssertTrue(paths inter dom fs' = {});
   AssertTrue(paths union dom fs' = dom fs);
   fs := fs');

end FileStoreTest
```

FS_DeleteFileDir Test

```
class DeleteFileDirTest is subclass of TestCase, FileSystemLayerObj


instance variables

protected
vals : FileSystemLayerValues;

protected
io : IO;


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   SetSystem(vals.sys1);
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (dcl status : FFS_Status;
   status := FS_DeleteFileDir(["etc","hosts"]);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := FS_DeleteFileDir(["bin","cp"]);
   AssertTrue(status = <FS_ErrorFileNotFound>);

   status := FS_DeleteFileDir(["etc"]);
   AssertTrue(status = <FS_ErrorFileStillOpen>);

   status := FS_DeleteFileDir(["bin"]);
   AssertTrue(status = <FS_ErrorDirectoryNonEmpty>);
  );

protected
TearDown: () ==> ()
```

```
TearDown() == skip;

end DeleteFileDirTest
```

Test Values

```
class FileSystemLayerValues is subclass of FileSystemLayerBase

instance variables

public
dirAttr  : Attributes
  := mk_Attributes(<Directory>);

public
fileAttr : Attributes
  := mk_Attributes(<RegularFile>);

public
dirInfo : FS_FileDirInfo
  := mk_FS_FileDirInfo(dirAttr);

public
fileInfo : FS_FileDirInfo
  := mk_FS_FileDirInfo(fileAttr);


public
table1 : OpenFilesTable
  := { 1 |-> mk_FS_OpenFileInfo(<Root>),
       2 |-> mk_FS_OpenFileInfo(["etc"]),
       3 |-> mk_FS_OpenFileInfo(["etc","resolv.conf"]),
       4 |-> mk_FS_OpenFileInfo(["etc","resolv.conf"]),
       5 |-> mk_FS_OpenFileInfo(["bin","wc"]) };

public
fs1 : FileStore
  := { <Root>  |-> mk_File(dirInfo),
       ["etc"] |-> mk_File(dirInfo),
       ["etc","conf.d"] |-> mk_File(dirInfo),
       ["etc","hosts"]  |-> mk_File(fileInfo),
       ["etc","resolv.conf"]  |-> mk_File(fileInfo),
       ["bin"] |-> mk_File(dirInfo),
       ["bin","ls"] |-> mk_File(fileInfo),
       ["bin","wc"] |-> mk_File(fileInfo) };

public
sys1 : System
  := mk_System(table1,fs1);

end FileSystemLayerValues
```

B.2.2 FS_OpenFileDir

Intel Test

```
class IntelTest


operations

public
Execute: () ==> ()
Execute() ==
  (dcl ts : TestSuite := new TestSuite();
       ts.AddTest(new FileStoreTest("FileStore Test"));
       ts.AddTest(new SystemTest("System Test"));
       ts.AddTest(new OpenFilesTableTest("OpenFilesTable Test"));
       ts.AddTest(new OpenFileDirTest("OpenFileDir Test"));

       ts.Run())

end IntelTest
```

System Test

```
class SystemTest is subclass of TestCase, FileSystemLayerOperations


instance variables

protected
vals : FileSystemLayerValues;

protected
sys : FileSystemLayerBase`System;

protected
io : IO;

protected
h : [FS_FileHandle];


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   sys  := mk_System({|->},{|->});
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (SimpleOpenTest(<Root>, vals.dirAttr, <FS_CreateAlways>);
```

```
   SimpleOpenTest(<Root>, vals.dirAttr, <FS_OpenAlways>);
   SimpleOpenTest(["etc"], vals.dirAttr, <FS_CreateNew>);
   SimpleOpenTest(["bin"], vals.dirAttr, <FS_CreateAlways>);
   SimpleOpenTest(["etc","hosts"], vals.fileAttr, <FS_OpenWrite>);
   SimpleOpenTest(["etc","conf.d"], vals.fileAttr, <FS_OpenRead>);
   SimpleOpenTest(["bin","ls"], vals.fileAttr, <FS_OpenWriteOnly>);
   SimpleOpenTest(["bin","wc"], vals.fileAttr, <FS_OpenWrite>);
   SimpleOpenTest(["etc", "resolv.conf"], vals.fileAttr, <FS_OpenWrite>));

protected
TearDown: () ==> ()
TearDown() == skip;

protected
SimpleOpenTest: Path * Attributes * FS_OpenMode ==> ()
SimpleOpenTest(path, attr, omode) ==
  let mk_(sys', handle) = FS_OpenFileDir_System(sys, path, attr, omode) in
  (AssertTrue(path in set dom sys'.fileStore);
   AssertTrue(sys'.fileStore(path).info.attributes = attr);
   if attr.fileType = <RegularFile>
   then (AssertTrue(handle <> nil);
         AssertTrue(path in set { ofi.path | ofi in set rng sys'.table });
         AssertTrue(dom sys'.table = dom sys.table union {handle});
         AssertTrue(card dom sys'.table = card dom sys.table + 1);
         AssertTrue(path = sys'.table(handle).path);
         AssertTrue(sys'.table(handle).accessMode = fs_open2access_mode_map(omode));
         AssertTrue(path in set dom sys'.fileStore);
         AssertTrue(sys'.fileStore(path).info.attributes = attr);
         AssertTrue(sys'.table(handle).fileOffset = getOpenOffset(sys'.fileStore(path),
             omode))
        )
   else AssertTrue(handle = nil);
   h := handle;
   sys := sys');

end SystemTest
```

File Store Test

```
class FileStoreTest is subclass of TestCase, FileSystemLayerOperations


instance variables

protected
vals : FileSystemLayerValues;

protected
fs : FileStore;

protected
io : IO;


operations
```

```
protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   fs   := {|->};
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (CreateOpenTest(<Root>, vals.dirAttr);
   CreateOpenTest(["etc"], vals.dirAttr);
   CreateOpenTest(["bin"], vals.dirAttr);
   CreateOpenTest(["etc", "hosts"], vals.fileAttr);
   CreateOpenTest(["etc", "conf.d"], vals.fileAttr);
   CreateOpenTest(["etc", "resolv.conf"], vals.fileAttr);
   CreateOpenTest(["bin", "ls"], vals.fileAttr);
   CreateOpenTest(["bin", "wc"], vals.fileAttr);

   DoNothingOpenTest(<Root>, vals.dirAttr);
   DoNothingOpenTest(["etc"], vals.dirAttr);
   DoNothingOpenTest(["bin"], vals.dirAttr);
   DoNothingOpenTest(["etc", "hosts"], vals.fileAttr);
   DoNothingOpenTest(["etc", "conf.d"], vals.fileAttr);
   DoNothingOpenTest(["etc", "resolv.conf"], vals.fileAttr);
   DoNothingOpenTest(["bin", "ls"], vals.fileAttr);
   DoNothingOpenTest(["bin", "wc"], vals.fileAttr));

protected
TearDown: () ==> ()
TearDown() == skip;

protected
CreateOpenTest: Path * Attributes ==> ()
CreateOpenTest(path,attr) ==
  let fs' = FS_OpenFileDir_FileStore(fs,path,attr) in
  (AssertTrue(path in set dom fs');
   AssertTrue(fs'(path).info.attributes = attr);
   fs := fs');

protected
DoNothingOpenTest: Path * Attributes ==> ()
DoNothingOpenTest(path,attr) ==
  let fs' = FS_OpenFileDir_FileStore(fs,path,attr) in
  (AssertTrue(fs = fs');
   fs := fs');

end FileStoreTest
```

Open Files Table Test

```
class OpenFilesTableTest is subclass of TestCase, FileSystemLayerOperations



instance variables

```

```
protected
vals : FileSystemLayerValues;

protected
table : OpenFilesTable;

protected
io : IO;

protected
h : [FS_FileHandle];


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals  := new FileSystemLayerValues();
   table := vals.table1;
   io    := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (dcl init         : OpenFilesTable := table;

   SimpleOpenTest(["bin"], <FS_CreateNew>, <Directory>);
   SimpleOpenTest(["etc","hosts"], <FS_CreateNew>, <RegularFile>);
   SimpleOpenTest(["etc","conf.d"], <FS_CreateNew>, <Directory>);
   SimpleOpenTest(<Root>, <FS_OpenRead>, <Directory>);
   SimpleOpenTest(["etc"], <FS_OpenRead>, <Directory>);
   SimpleOpenTest(["bin","ls"], <FS_CreateNew>, <RegularFile>);

   AssertTrue(card dom table = card dom init + 2));

protected
TearDown: () ==> ()
TearDown() == skip;

protected
SimpleOpenTest: Path * FS_OpenMode * FileType ==> ()
SimpleOpenTest(path,omode, fileType) ==
  let mk_(table',handle) = FS_OpenFileDir_Table(table, path, omode, fileType) in
  (if fileType = <RegularFile>
   then (AssertTrue(handle <> nil);
         AssertTrue(path in set { ofi.path | ofi in set rng table' });
         AssertTrue(dom table' = dom table union {handle});
         AssertTrue(card dom table' = card dom table + 1);
         AssertTrue(path = table'(handle).path);
         AssertTrue(table'(handle).accessMode = fs_open2access_mode_map(omode)))
   else AssertTrue(handle = nil);

   table := table';
   h     := handle);

end OpenFilesTableTest
```

FS_OpenFileDir Test

```
class OpenFileDirTest is subclass of TestCase, FileSystemLayerObject


instance variables

protected
vals : FileSystemLayerValues;

protected
io : IO;


operations

protected
SetUp: () ==> ()
SetUp() ==
  (vals := new FileSystemLayerValues();
   SetSystem(vals.sys1);
   io   := new IO());

protected
RunTest: () ==> ()
RunTest() ==
  (dcl status : FFS_Status;

   status := SimpleOpenTest(["etc","hosts"], vals.fileAttr, <FS_OpenAlways>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(<Root>, vals.dirAttr, <FS_CreateAlways>);
   AssertTrue(status = <FS_ErrorDirectoryNonEmpty>);

   status := SimpleOpenTest(["bin","ls"], vals.fileAttr, <FS_CreateAlways>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(["bin","wc"], vals.fileAttr, <FS_OpenRead>);
   AssertTrue(status = <FFS_StatusSuccess>);

   status := SimpleOpenTest(["bin"], vals.dirAttr, <FS_OpenRead>);
   AssertTrue(status = <FFS_StatusInvalidParameter>);

   status := SimpleOpenTest(["bin"], vals.dirAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorFileAlreadyExists>);

   status := SimpleOpenTest(["etc","resolv.conf"], vals.fileAttr, <FS_CreateAlways>);
   AssertTrue(status = <FS_ErrorFileStillOpen>);

   status := SimpleOpenTest(["usr"], vals.dirAttr, <FS_OpenRead>);
   AssertTrue(status = <FS_ErrorFileNotFound>);

   status := SimpleOpenTest(["usr"], vals.dirAttr, <FS_OpenWrite>);
   AssertTrue(status = <FS_ErrorFileNotFound>);

   status := SimpleOpenTest(["usr","share"], vals.dirAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorInvalidPath>);
```

```
   status := SimpleOpenTest(["bin","ls","somefile"], vals.fileAttr, <FS_CreateNew>);
   AssertTrue(status = <FS_ErrorInvalidPath>));

protected
TearDown: () ==> ()
TearDown() == skip;

protected
SimpleOpenTest: Path * Attributes * FS_OpenMode ==> FFS_Status
SimpleOpenTest(path, attr, omode) ==
  let mk_(handle,status) = FS_OpenFileDir(path, attr, omode) in
  (if (attr.fileType = <RegularFile> and status = <FFS_StatusSuccess>)
   then (AssertTrue(handle <> nil);
         AssertTrue(sys.table(handle).fileOffset = getOpenOffset(sys.fileStore(path),
             omode)))
   else AssertTrue(handle = nil);
   return status);

end OpenFileDirTest
```

Test Values

```
class FileSystemLayerValues is subclass of FileSystemLayerBase

instance variables

public
dirAttr  : Attributes
  := mk_Attributes(<Directory>);

public
fileAttr : Attributes
  := mk_Attributes(<RegularFile>);

public
dirInfo : FS_FileDirInfo
  := mk_FS_FileDirInfo(dirAttr);

public
fileInfo : FS_FileDirInfo
  := mk_FS_FileDirInfo(fileAttr);


public
table1 : OpenFilesTable
  := { 3 |-> mk_FS_OpenFileInfo(1, <FS_AccessReadWrite>, ["etc","resolv.conf"]),
       4 |-> mk_FS_OpenFileInfo(1, <FS_AccessReadOnly>,  ["etc","resolv.conf"]),
       5 |-> mk_FS_OpenFileInfo(1, <FS_AccessReadWrite>, ["bin","wc"]) };

public
fs1 : FileStore
  := { <Root>  |-> mk_File(dirInfo, nil),
       ["etc"] |-> mk_File(dirInfo, nil),
       ["etc","conf.d"] |-> mk_File(dirInfo, nil),
       ["etc","hosts"]  |-> mk_File(fileInfo, [mk_token(1),mk_token(2)]),
       ["etc","resolv.conf"]  |-> mk_File(fileInfo, [mk_token(3)]),
```

```
       ["bin"] |-> mk_File(dirInfo, nil),
       ["bin","ls"] |-> mk_File(fileInfo, [mk_token(4)]),
       ["bin","wc"] |-> mk_File(fileInfo, [mk_token(5)]) };

public
sys1 : System
  := mk_System(table1,fs1);

end FileSystemLayerValues
```

B.3 Alloy Model

B.3.1 Flash File System Core

```
module FlashFileSystemCore
open util/integer
open util/boolean


abstract sig FFS_Status {}

one sig FFS_StatusSuccess                extends FFS_Status {}
one sig FS_ErrorFileNotFound             extends FFS_Status {}
one sig FS_ErrorFileStillOpen            extends FFS_Status {}
one sig FS_ErrorDirectoryNonEmpty        extends FFS_Status {}
one sig FS_ErrorFileAlreadyExists        extends FFS_Status {}
one sig FS_ErrorInvalidPath              extends FFS_Status {}
one sig FFS_StatusInvalidParameter       extends FFS_Status {}
one sig FFS_StatusUnknown                extends FFS_Status {}
```

File System Layer Base

```
module FileSystemLayerBase
open FlashFileSystemCore
open RelCalc


sig System {
  table    : OpenFilesTable,
  fileStore: FileStore
}

pred SystemInvariantVDM[sys: System] {
  OpenFilesTableInvariantVDM[sys.table] and
  FileStoreInvariantVDM[sys.fileStore]  and
  SystemInvariant[sys]
}

pred SystemInvariant[sys: System]{
  let oft = sys.table.map,
      fs  = sys.fileStore.map {
    RelCalc/rng[oft.path] in RelCalc/dom[fs] and
    (oft.path).(fs).(info.attributes.fileType) in (FS_FileHandle->RegularFile)
  }
}


sig FileStore {
  map: Path -> File
}

pred FileStoreInvariantVDM[fs: FileStore] {
  RelCalc/Simple[fs.map, File]           and
  RelCalc/Injective[fs.map, Path]        and
  PathInvariantVDM[RelCalc/dom[fs.map]]  and
```

```
  FileInvariantVDM[RelCalc/rng[fs.map]] and
  FileStoreInvariant[fs]
}

pred FileStoreInvariant[fs: FileStore] {
  (fs.map).(File->Directory) in dirName.(fs.map).info.attributes.fileType
}


abstract sig Path {
  dirName: Path
}

sig FileNames extends Path {}
one sig Root extends Path {}

pred PathInvariantVDM[path : Path]{
  dirNameProperties
}

pred dirNameProperties {
  RelCalc/Function[dirName,Path,Path]                and
  RelCalc/Reflexive[(RelCalc/id[Root]).dirName,Root] and
  RelCalc/Acyclic[(RelCalc/id[FileNames]).dirName,FileNames]
}


sig File {
  info     : FS_FileDirInfo,
  contents : OptionalFileContents
}

pred FileInvariantVDM[f: File]{
  OptionalFileContentsInvariantVDM[f.contents] and
  all file: f {
    file.contents not in NilFileContents
    => let fileContents = file.contents.(^nextChunk) |
       no (fileContents & (File.contents.(^nextChunk) - fileContents))
  }
  FileInvariant[f]
}

pred FileInvariant[f: File] {   -
  all file: f {
    (file.info.attributes.fileType in Directory and
     file.contents in NilFileContents)
    or
    (file.info.attributes.fileType in RegularFile and
     file.contents in FileContents)
  }
}


sig FS_FileDirInfo {
  attributes     : Attributes
}
```

```
sig Attributes {
  fileType: FileType
}


abstract sig FileType {}

one sig RegularFile extends FileType {}
one sig Directory extends FileType {}


abstract sig OptionalFileContents {}

one sig NilFileContents extends OptionalFileContents {}

pred OptionalFileContentsInvariantVDM[ofc: OptionalFileContents] {
  FileContentsInvariantVDM[(ofc & FileContents)]
}

abstract sig FileContents extends OptionalFileContents {}

sig Chunk extends FileContents {
  nextChunk: FileContents
}

one sig Nothing extends FileContents {}

pred FileContentsInvariantVDM[cont: FileContents] {
  RelCalc/Function[nextChunk,Chunk,FileContents] and
  RelCalc/Acyclic[(RelCalc/id[Chunk]).nextChunk,Chunk]
}


sig OpenFilesTable {
  map: FS_FileHandle -> FS_OpenFileInfo,
}

pred OpenFilesTableInvariantVDM[table: OpenFilesTable] {
  RelCalc/Simple[table.map,FS_OpenFileInfo]           and
  RelCalc/Injective[table.map, FS_FileHandle]         and
  FS_OpenFileInfoInvariantVDM[RelCalc/rng[table.map]]
}


sig FS_FileHandle extends OptionalFileHandle {}

abstract sig OptionalFileHandle {}
one sig NilFileHandle extends OptionalFileHandle {}


sig FS_OpenFileInfo {
  fileOffset:  FileContents,
  accessMode:  FS_AccessMode,
  path      :  Path
}

pred FS_OpenFileInfoInvariantVDM[ofi: FS_OpenFileInfo]{
  FileContentsInvariantVDM[ofi.fileOffset]       and
```

```
  PathInvariantVDM[ofi.path]
}


abstract sig FS_OpenMode {}

one sig FS_CreateNew             extends FS_OpenMode {}
one sig FS_CreateAlways          extends FS_OpenMode {}
one sig FS_OpenRead              extends FS_OpenMode {}
one sig FS_OpenWrite             extends FS_OpenMode {}
one sig FS_OpenAlways            extends FS_OpenMode {}
one sig FS_OpenWriteOnly         extends FS_OpenMode {}
one sig FS_CreateAlwaysReadOnly extends FS_OpenMode {}
one sig FS_CreateNewReadOnly    extends FS_OpenMode {}


abstract sig FS_AccessMode {}

one sig  FS_AccessReadOnly  extends FS_AccessMode {}
one sig  FS_AccessWriteOnly extends FS_AccessMode {}
one sig  FS_AccessReadWrite extends FS_AccessMode {}


pred fs_open2access_mode_map[omode: FS_OpenMode, amode: FS_AccessMode] {
  (omode = FS_CreateNew            and amode = FS_AccessReadWrite) or
  (omode = FS_CreateAlways         and amode = FS_AccessReadWrite) or
  (omode = FS_OpenRead             and amode = FS_AccessReadOnly)  or
  (omode = FS_OpenWrite            and amode = FS_AccessReadWrite) or
  (omode = FS_OpenAlways           and amode = FS_AccessReadWrite) or
  (omode = FS_OpenWriteOnly        and amode = FS_AccessWriteOnly) or
  (omode = FS_CreateAlwaysReadOnly and amode = FS_AccessReadOnly)  or
  (omode = FS_CreateNewReadOnly    and amode = FS_AccessReadOnly)
}

pred isDirectory[info: FS_FileDirInfo] {
  info.attributes.fileType in Directory
}

pred isRegularFile[info: FS_FileDirInfo] {
  info.attributes.fileType in RegularFile
}

pred hasSubFiles[fs: FileStore, path: Path] {
  path in RelCalc/dom[fs.map].dirName
}

pred isRoot[path: Path] {
  path in Root
}

pred isElemFileStore[path: Path, fs: FileStore] {
  path in RelCalc/dom[fs.map]
}

pred isElemFileStore[file: File, fs: FileStore] {
  file in RelCalc/rng[fs.map]
}
```

```
pred isElemTablePath[p: Path, table: OpenFilesTable] {
  p in RelCalc/rng[table.map].path
}

pred isElemTableHandle[handle: FS_FileHandle, table: OpenFilesTable] {
  handle in RelCalc/dom[table.map]
}

pred isElemTable[ofi: FS_OpenFileInfo, table: OpenFilesTable] {
  ofi in RelCalc/rng[table.map]
}

pred isDirName[parent, path: Path] {
  parent = path.dirName
}

pred ShowFileStore[fs: FileStore] {
  some fs.map                                              and
  Path = RelCalc/dom[fs.map]                               and
  File = RelCalc/rng[fs.map]                               and
  FS_FileDirInfo = RelCalc/rng[fs.map].info                and
  Attributes = RelCalc/rng[fs.map].info.attributes         and
  FileType = RelCalc/rng[fs.map].info.attributes.fileType  and
  let fc = RelCalc/rng[fs.map].contents - NilFileContents |
    FileContents = fc.(*nextChunk)                         and
  #(Nothing.~nextChunk) > 2
  #(File.contents - (Nothing + NilFileContents)) > 2
  #RelCalc/rng[contents.nextChunk] > 2
  #FileNames.dirName > 2
  FileStoreInvariantVDM[fs]
}
run ShowFileStore for 7 but 1 FileStore,
                            0 System,
                            0 OpenFilesTable,
                            0 FS_FileHandle,
                            0 FS_OpenFileInfo


pred ShowOpenFilesTable[table: OpenFilesTable] {
  some table.map                              and
  FS_FileHandle = RelCalc/dom[table.map]      and
  FS_OpenFileInfo = RelCalc/rng[table.map]    and
  OpenFilesTableInvariantVDM[table]
}
run ShowOpenFilesTable for 3 but 1 OpenFilesTable,
                                 0 System,
                                 0 FileStore,
                                 0 File


pred SystemShowConstraints[sys: System] {
  Path = RelCalc/dom[sys.fileStore.map]          and
  File = RelCalc/rng[sys.fileStore.map]          and
  FS_FileHandle = RelCalc/dom[sys.table.map]     and
  FS_OpenFileInfo = RelCalc/rng[sys.table.map]
}
```

```
assert DirectoryFileOffset {
  all sys: System, ofi: RelCalc/rng[sys.table.map] {
    SystemInvariantVDM[sys]    =>
    ((ofi.path).(sys.fileStore.map).info.attributes.fileType = Directory =>
ofi.fileOffset = Nothing)
  }
}
check DirectoryFileOffset for 8


pred ShowSystem[sys: System] {
  ShowFileStore[sys.fileStore]  and
  ShowOpenFilesTable[sys.table] and
  OpenFilesTable = sys.table    and
  FileStore = sys.fileStore     and
  SystemInvariantVDM[sys]
}
run ShowSystem for 7 but 1 System
```

File System Layer Operations

```
module FileSystemLayerOperations
open RelCalc
open FlashFileSystemCore
open FileSystemLayerBase


pred FS_DeleteFileDir_Main[sys, sys': System, full_path:Path, status: FFS_Status] {
  (isElemFileStore[full_path, sys.fileStore] and
   pre_FS_DeleteFileDir_System[sys,full_path])
  =>   (FS_DeleteFileDir_System[sys,sys',full_path] and
        status = FFS_StatusSuccess)
  else (FS_DeleteFileDir_Exception[sys,full_path,status] and
        sys' = sys)
}


pred FS_DeleteFileDir_System[sys, sys': System, full_path: Path] {
  FS_DeleteFileDir_FileStore[sys.fileStore,sys'.fileStore,{full_path}] and
  sys.table = sys'.table
}

pred pre_FS_DeleteFileDir_System[sys: System, full_path: Path] {
  not isElemTablePath[full_path,sys.table]                  and
  pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
}


pred FS_DeleteFileDir_FileStore[fs,fs': FileStore, paths: set Path] {
  fs'.map = fs.map - (paths -> (paths.(fs.map)))
}

pred pre_FS_DeleteFileDir_FileStore[fs: FileStore, paths: set Path] {
  (((Path - paths)->Path) & iden).(fs.map) in dirName.((Path - paths)->File)
}
```

```
pred FS_DeleteFileDir_Exception[sys: System, full_path: Path, status: FFS_Status] {
  not isElemFileStore[full_path,sys.fileStore]
  =>   status = FS_ErrorFileNotFound
  else (isElemTablePath[full_path,sys.table]
       =>   status = FS_ErrorFileStillOpen
       else ((isDirectory[(sys.fileStore.map[full_path]).info] and
             hasSubFiles[sys.fileStore,full_path])
            =>   status = FS_ErrorDirectoryNonEmpty
            else status = FFS_StatusUnknown))
}


pred FS_OpenFileDir_Main[sys,sys'  : System,
                         full_path : Path,
                         attributes: Attributes,
                         omode     : FS_OpenMode,
                         handle    : OptionalFileHandle,
                         status    : FFS_Status         ] {
  (pre_FS_OpenFileDir_System[sys,full_path,attributes,omode] and
   checkOpenMode[sys,full_path,omode])
  =>   (mustDeleteFirst[sys.fileStore,full_path,omode] and
        (full_path = Root => attributes.fileType = Directory))
       =>   (some dstatus: FFS_Status, dsys: System {
               FS_DeleteFileDir_Main[sys,dsys,full_path,dstatus] and
               dstatus = FFS_StatusSuccess
               =>   (FS_OpenFileDir_System[dsys,sys',full_path,attributes,omode,handle]
                   and
                     status = FFS_StatusSuccess)
               else (sys'   = sys           and
                     handle = NilFileHandle and
                     status = dstatus)
            })
       else (FS_OpenFileDir_System[sys,sys',full_path,attributes,omode,handle] and
             status = FFS_StatusSuccess)
  else (sys'   = sys           and
        handle = NilFileHandle and
        FS_OpenFileDir_Exception[sys,full_path,omode,status])
}


pred FS_OpenFileDir_System[sys,sys' : System,
                           full_path: Path,
                           attr     : Attributes,
                           omode    : FS_OpenMode,
                           handle   : OptionalFileHandle] {
  FS_OpenFileDir_FileStore[sys.fileStore,sys'.fileStore,full_path,attr] and
  let fileType = full_path.(sys'.fileStore.map).info.attributes.fileType {
    FS_OpenFileDir_Table[sys.table,sys'.table,full_path,omode,fileType,handle]
  }
}

pred pre_FS_OpenFileDir_System[sys      : System,
                               full_path: Path,
                               attr     : Attributes,
                               omode    : FS_OpenMode] {
  pre_FS_OpenFileDir_FileStore[sys.fileStore,full_path,attr]
}
```

```
pred FS_OpenFileDir_FileStore[fs,fs'   : FileStore,
                              full_path: Path,
                              attr     : Attributes] {
  not isElemFileStore[full_path,fs]
  =>   (one file: File {
          fs'.map = fs.map + (full_path -> file)                 and
          file.info.attributes = attr                            and
          (attr.fileType in Directory => file.contents in NilFileContents) and
          (attr.fileType in RegularFile => file.contents in FileContents) and
          not isElemFileStore[file,fs']
       })
  else fs'.map = fs.map
}

pred pre_FS_OpenFileDir_FileStore[fs       : FileStore,
                                  full_path: Path,
                                  attr     : Attributes] {
  (full_path = Root and
   attr.fileType = Directory)
  or
  (isElemFileStore[full_path.dirName,fs] and
   isDirectory[(fs.map[full_path.dirName]).info])
}


pred FS_OpenFileDir_Table[table,table': OpenFilesTable,
                          full_path   : Path,
                          omode       : FS_OpenMode,
                          fileType    : FileType,
                          handle      : OptionalFileHandle] {
  fileType in Directory
  =>   (table'.map = table.map and
        handle = NilFileHandle)
  else one ofi: FS_OpenFileInfo {
         not isElemTableHandle[handle,table]           and
         not isElemTable[ofi,table]                    and
         ofi.fileOffset = Nothing                      and
         ofi.path = full_path                          and
         fs_open2access_mode_map[omode,ofi.accessMode] and
         table'.map = table.map + (handle -> ofi)
       }
}


pred FS_OpenFileDir_Exception[sys      : System,
                              full_path: Path,
                              omode    : FS_OpenMode,
                              status   : FFS_Status] {
  (isCreateNew[omode] and isElemFileStore[full_path,sys.fileStore])
  =>   status = FS_ErrorFileAlreadyExists
  else (isCreateAlways[omode] and isElemTablePath[full_path,sys.table])
       =>   status = FS_ErrorFileStillOpen
       else (isOpen[omode] and not isElemFileStore[full_path,sys.fileStore])
            =>   status = FS_ErrorFileNotFound
            else (isOpen[omode] and isElemFileStore[full_path, sys.fileStore]
                                and not isRegularFile[sys.fileStore.map[full_path].info
```

```
                                      ])
                  => status = FFS_StatusInvalidParameter
                  else not (full_path in Root or
                            isElemFileStore[full_path.dirName,sys.fileStore])
                       =>   status = FS_ErrorInvalidPath
                       else not (full_path in Root or
                                 isDirectory[(sys.fileStore.map[full_path.dirName]).info
                                     ])
                            =>   status = FS_ErrorInvalidPath
                            else status = FFS_StatusUnknown
}


pred mustDeleteFirst[fs: FileStore, path: Path, omode: FS_OpenMode] {
  isCreateAlways[omode] and isElemFileStore[path,fs]
}

pred checkOpenMode[sys  : System,
                   path : Path,
                   omode: FS_OpenMode] {
  not (isCreateNew[omode]    and isElemFileStore[path,sys.fileStore])     and
  not (isCreateAlways[omode] and isElemTablePath[path,sys.table])         and
  not (isOpen[omode]         and not isElemFileStore[path,sys.fileStore])
}

pred isCreateAlways[omode: FS_OpenMode] {
  omode in (FS_CreateAlways + FS_CreateNewReadOnly)
}

pred isCreateNew[omode: FS_OpenMode] {
  omode in (FS_CreateNew + FS_CreateNewReadOnly)
}

pred isOpen[omode: FS_OpenMode] {
  omode in (FS_OpenRead + FS_OpenWrite + FS_OpenWriteOnly)
}

pred getOpenOffset[file: File, omode: FS_OpenMode, contents: FileContents] {
  (omode in FS_OpenWrite + FS_OpenAlways and contents = getLastChunk[file]) or
  (contents = Nothing)
}

fun getLastChunk[file: File] : FileContents {
  file.contents in Nothing
  =>   Nothing
  else RelCalc/dom[(file.contents.(*nextChunk)->Nothing) & nextChunk]
}


pred FS_Init_Main[sys: System, status: FFS_Status] {
  FS_Init_System[sys]       and
  status = FFS_StatusSuccess
}

pred FS_Init_System[sys: System] {
  FS_Init_FileStore[sys.fileStore] and
  FS_Init_Table[sys.table]
}
```

```
pred FS_Init_Table[table: OpenFilesTable] {
  no table.map
}

pred FS_Init_FileStore[fs: FileStore] {
  some file: File, fdi: FS_FileDirInfo, attr: Attributes {
    PathInvariantVDM[Root]  and
    FileInvariantVDM[file]
    => attr.fileType = Directory and
       fdi.attributes = attr     and
       file.info = fdi           and
       fs.map = Root -> file
  }
}


pred ShowFS_DeleteFileDir_Main[sys,sys': System, full_path: Path, status: FFS_Status] {
  OpenFilesTable = sys.table + sys'.table                                 and
  File = RelCalc/rng[sys.fileStore.map] + RelCalc/rng[sys'.fileStore.map] and
  SystemInvariantVDM[sys]                                                 and
  FS_DeleteFileDir_Main[sys,sys',full_path,status]                        and
  sys not = sys'
}
ShowDeleteFileDir_Main: run ShowFS_DeleteFileDir_Main
                        for 3 but 2 System,
                                  1 FFS_Status


pred ShowFS_DeleteFileDir_System[sys,sys': System, full_path: Path] {
  OpenFilesTable = sys.table + sys'.table                                 and
  Path = RelCalc/dom[sys.fileStore.map] + RelCalc/dom[sys'.fileStore.map] and
  File = RelCalc/rng[sys.fileStore.map] + RelCalc/rng[sys'.fileStore.map] and
  SystemInvariantVDM[sys]                                                 and
  pre_FS_DeleteFileDir_System[sys,full_path]                              and
  FS_DeleteFileDir_System[sys,sys',full_path]                             and
  sys not = sys'
}
ShowDeleteFileDir_System: run ShowFS_DeleteFileDir_System
                          for 3 but 2 System


pred ShowFS_DeleteFileDir_FileStore[fs,fs': FileStore,
                                    paths: set Path] {
  Path = RelCalc/dom[fs.map] + RelCalc/dom[fs'.map] and
  File = RelCalc/rng[fs.map] + RelCalc/rng[fs'.map] and
  some paths                                        and
  some fs.map                                       and
  some fs'.map                                      and
  fs not = fs'                                      and
  FileStoreInvariantVDM[fs]                         and
  PathInvariantVDM[Path]                            and
  pre_FS_DeleteFileDir_FileStore[fs,paths]          and
  FS_DeleteFileDir_FileStore[fs,fs',paths]
}
ShowDeleteFileDir_FileStore: run ShowFS_DeleteFileDir_FileStore
                             for 3 but 2 FileStore,
                                       0 System,
```

```
                                    0 OpenFilesTable


pred ShowFS_DeleteFileDir_Exception[sys      : System,
                                    full_path: Path,
                                    status   : FFS_Status] {
  SystemInvariantVDM[sys]     and
  PathInvariantVDM[full_path] and
  FS_DeleteFileDir_Exception[sys,full_path,status]
}

pred ShowFS_DeleteFileDir_ErrorFileNotFound[sys      : System,
                                            full_path: Path,
                                            status   : FFS_Status] {
  ShowFS_DeleteFileDir_Exception[sys,full_path,status] and
  status = FS_ErrorFileNotFound
}
Show_Delete_FileNotFound: run ShowFS_DeleteFileDir_ErrorFileNotFound
                          for 3 but 1 System,
                                    1 FFS_Status


pred ShowFS_DeleteFileDir_ErrorFileStillOpen[sys: System, full_path: Path,status:
    FFS_Status] {
  ShowFS_DeleteFileDir_Exception[sys,full_path,status] and
  status = FS_ErrorFileStillOpen
}
Show_Delete_FileStillOpen: run ShowFS_DeleteFileDir_ErrorFileStillOpen
                           for 3 but 1 System,
                                     1 FFS_Status


pred ShowFS_DeleteFileDir_ErrorDirectoryNonEmpty[sys: System, full_path: Path,status:
    FFS_Status] {
  ShowFS_DeleteFileDir_Exception[sys,full_path,status] and
  status = FS_ErrorDirectoryNonEmpty
}
Show_Delete_DirectoryNonEmpty: run ShowFS_DeleteFileDir_ErrorDirectoryNonEmpty
                               for 3 but 1 System,
                                         1 FFS_Status


pred Show_FS_Init_Main[sys: System] {
  SystemInvariantVDM[sys]  and
  FS_Init_Main[sys, FFS_StatusSuccess]
}
Show_Init_Main: run Show_FS_Init_Main
                for 3 but 1 System,
                          1 FFS_Status


pred ShowFS_OpenFileDir_Main[sys      : System,
                             full_path: Path,
                             attr     : Attributes,
                             omode    : FS_OpenMode,
                             sys'     : System,
                             handle   : OptionalFileHandle,
                             status   : FFS_Status         ] {
```

```
  SystemInvariantVDM[sys]                                              and
  PathInvariantVDM[Path]                                               and
  some full_path                                                       and
  some omode                                                           and
  some sys.table.map                                                   and
  some sys'.table.map                                                  and
  sys.table not = sys'.table                                           and
  full_path not in RelCalc/rng[sys.table.map].path                     and
  some attr                                                            and
  some sys.fileStore.map                                               and
  some sys'.fileStore.map                                              and
  sys.fileStore not = sys'.fileStore                                   and
  FS_OpenFileDir_Main[sys,sys',full_path,attr,omode,handle,status]
}
ShowOpenFileDir_Main: run ShowFS_OpenFileDir_Main
                      for 3 but 2 System


pred ShowFS_OpenFileDir_System[sys      : System,
                               full_path: Path,
                               attr     : Attributes,
                               omode    : FS_OpenMode,
                               sys'     : System,
                               handle   : OptionalFileHandle] {
  SystemInvariantVDM[sys]                                              and
  PathInvariantVDM[Path]                                               and
  some full_path                                                       and
  some omode                                                           and
  some sys.table.map                                                   and
  some sys'.table.map                                                  and
  sys.table not = sys'.table                                           and
  full_path not in RelCalc/rng[sys.table.map].path                     and
  some attr                                                            and
  some sys.fileStore.map                                               and
  some sys'.fileStore.map                                              and
  sys.fileStore not = sys'.fileStore                                   and
  pre_FS_OpenFileDir_System[sys,full_path,attr,omode]                  and
  FS_OpenFileDir_System[sys,sys',full_path,attr,omode,handle]
}
ShowOpenFileDir_System: run ShowFS_OpenFileDir_System
                        for 3 but 2 System


pred ShowFS_OpenFileDir_FileStore[fs       : FileStore,
                                  full_path: Path,
                                  attr     : Attributes,
                                  fs'      : FileStore  ] {
  Path = RelCalc/dom[fs.map] + RelCalc/dom[fs'.map] and
  some full_path                                    and
  some attr                                         and
  some fs.map                                       and
  some fs'.map                                      and
  fs not = fs'                                      and
  FileStoreInvariantVDM[fs]                         and
  PathInvariantVDM[Path]                            and
  pre_FS_OpenFileDir_FileStore[fs,full_path,attr]   and
  FS_OpenFileDir_FileStore[fs,fs',full_path,attr]
}
```

```
ShowOpenFileDir_FileStore: run ShowFS_OpenFileDir_FileStore
                           for 3 but 2 FileStore,
                                     0 System,
                                     0 OpenFilesTable


pred ShowFS_OpenFileDir_Table[table    : OpenFilesTable,
                              full_path: Path,
                              omode    : FS_OpenMode,
                              fileType : FileType,
                              table'   : OpenFilesTable,
                              handle   : OptionalFileHandle  ] {
  OpenFilesTableInvariantVDM[table]                     and
  PathInvariantVDM[full_path]                           and
  some full_path                                        and
  some omode                                            and
  some table.map                                        and
  some table'.map                                       and
  table not = table'                                    and
  full_path not in RelCalc/rng[table.map].path          and
  FS_FileHandle = RelCalc/dom[table.map + table'.map]   and
  FS_OpenFileInfo = RelCalc/rng[table.map + table'.map] and
  fileType = Directory                                  and
  handle = NilFileHandle                                and
  FS_OpenFileDir_Table[table,table',full_path,omode,fileType,handle]
}
ShowOpenFileDir_Table: run ShowFS_OpenFileDir_Table
                       for 5 but 0 FileStore,
                                 0 System,
                                 2 OpenFilesTable


pred ShowFS_OpenFileDir_Exception[sys      : System,
                                  full_path: Path,
                                  omode    : FS_OpenMode,
                                  status   : FFS_Status  ] {
  SystemInvariantVDM[sys]     and
  PathInvariantVDM[full_path] and
  FS_OpenFileDir_Exception[sys,full_path,omode,status]
}

pred ShowFS_OpenFileDir_ErrorInvalidPath_1[sys      : System,
                                           full_path: Path,
                                           omode    : FS_OpenMode,
                                           status   : FFS_Status  ] {
  ShowFS_OpenFileDir_Exception[sys,full_path,omode,status] and
  not isElemFileStore[full_path.dirName,sys.fileStore]     and
  status = FS_ErrorInvalidPath
}
Show_Open_InvalidPath1: run ShowFS_OpenFileDir_ErrorInvalidPath_1
                  for 3 but 1 System,
                            1 FFS_Status


pred ShowFS_OpenFileDir_ErrorInvalidPath_2[sys      : System,
                                           full_path: Path,
                                           omode    : FS_OpenMode,
                                           status   : FFS_Status  ] {
```

```
  ShowFS_OpenFileDir_Exception[sys,full_path,omode,status] and
  not isDirectory[(sys.fileStore.map[full_path.dirName]).info]    and
  status = FS_ErrorInvalidPath
}
Show_Open_InvalidPath2: run ShowFS_OpenFileDir_ErrorInvalidPath_2
                  for 3 but 1 System,
                            1 FileStore,
                            1 OpenFilesTable,
                            1 FFS_Status


pred ShowFS_OpenFileDir_ErrorFileStillOpen[sys      : System,
                                           full_path: Path,
                                           omode    : FS_OpenMode,
                                           status   : FFS_Status  ] {
  ShowFS_OpenFileDir_Exception[sys,full_path,omode,status] and
  isElemTablePath[full_path,sys.table]                     and
  status = FS_ErrorFileStillOpen
}
Show_Open_FileStillOpen: run ShowFS_OpenFileDir_ErrorFileStillOpen
                         for 3 but 1 System,
                                   1 FileStore,
                                   1 OpenFilesTable,
                                   1 FFS_Status
```

B.4 Model Checking

B.4.1 FS_DeleteFileDir

```
module OperationsModelCheck
open RelCalc
open FileSystemLayerOperations


assert Delete_FileStore {
  all fs,fs': FileStore, paths: set Path {
    FileStoreInvariantVDM[fs]                  and
    PathInvariantVDM[paths]                    and
    pre_FS_DeleteFileDir_FileStore[fs,paths] and
    FS_DeleteFileDir_FileStore[fs,fs',paths]
    => RelCalc/dom[fs'.map] = RelCalc/dom[fs.map] - paths and
       FileStoreInvariantVDM[fs']
  }
}
Check_Delete_FileStore: check Delete_FileStore


assert Delete_System {
  all sys, sys': System, full_path: Path {
    SystemInvariantVDM[sys]                     and
    PathInvariantVDM[full_path]                 and
    pre_FS_DeleteFileDir_System[sys,full_path] and
    FS_DeleteFileDir_System[sys,sys',full_path]
    => full_path not in RelCalc/dom[sys'.fileStore.map] and
       sys.table = sys'.table                           and
       SystemInvariantVDM[sys']                         and
       RelCalc/dom[sys'.fileStore.map] = RelCalc/dom[sys.fileStore.map] - full_path
  }
}
Check_Delete_System: check Delete_System


assert Delete_Exception_StatusUnknown {
  all sys: System, full_path: Path, status: FFS_StatusUnknown {
    SystemInvariantVDM[sys]                                 and
    PathInvariantVDM[full_path]                             and
    not (full_path in RelCalc/dom[sys.fileStore.map] and
         pre_FS_DeleteFileDir_System[sys,full_path]      ) and
    FS_DeleteFileDir_Exception[sys,full_path,status]
    => not status = FFS_StatusUnknown
  }
}
Check_Delete_Exception_StatusUnknown: check Delete_Exception_StatusUnknown


assert Delete_Exception_FileNotFound {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                     and
    PathInvariantVDM[full_path]                 and
    not isElemFileStore[full_path,sys.fileStore]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileNotFound]
  }
```

```
}
Check_Delete_Exception_FileNotFound: check Delete_Exception_FileNotFound


assert Delete_Exception_FileStillOpen {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                        and
    PathInvariantVDM[full_path]                    and
    isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileStillOpen]
  }
}
Check_Delete_Exception_FileStillOpen: check Delete_Exception_FileStillOpen


assert Delete_Exception_DirectoryNonEmpty {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                              and
    PathInvariantVDM[full_path]                          and
    isDirectory[(full_path.(sys.fileStore.map)).info] and
    hasSubFiles[sys.fileStore,full_path]                 and
    not isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorDirectoryNonEmpty]
  }
}
Check_Delete_Exception_DirectoryNonEmpty: check Delete_Exception_DirectoryNonEmpty


assert Delete_Main_Success {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                       and
    PathInvariantVDM[full_path]                   and
    full_path in RelCalc/dom[sys.fileStore.map] and
    pre_FS_DeleteFileDir_System[sys,full_path]  and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys'] and
       status = FFS_StatusSuccess
  }
}
Check_Delete_Main_Success: check Delete_Main_Success


assert Delete_Main_Error {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                             and
    PathInvariantVDM[full_path]                         and
    not (full_path in RelCalc/dom[sys.fileStore.map] and
         pre_FS_DeleteFileDir_System[sys,full_path]) and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys']       and
       status not = FFS_StatusSuccess
  }
}
Check_Delete_Main_Error: check Delete_Main_Error
```

B.4.2 FS_OpenFileDir

```
module OperationsModelCheck
open RelCalc
open FileSystemLayerOperations


assert Delete_FileStore {
  all fs,fs': FileStore, paths: set Path {
    FileStoreInvariantVDM[fs]                 and
    PathInvariantVDM[paths]                   and
    pre_FS_DeleteFileDir_FileStore[fs,paths] and
    FS_DeleteFileDir_FileStore[fs,fs',paths]
    => RelCalc/dom[fs'.map] = RelCalc/dom[fs.map] - paths and
       FileStoreInvariantVDM[fs']

  }
}
Check_Delete_FileStore: check Delete_FileStore


assert Delete_System {
  all sys, sys': System, full_path: Path {
    SystemInvariantVDM[sys]                     and
    PathInvariantVDM[full_path]                 and
    pre_FS_DeleteFileDir_System[sys,full_path] and
    FS_DeleteFileDir_System[sys,sys',full_path]
    => full_path not in RelCalc/dom[sys'.fileStore.map] and
       sys.table = sys'.table                           and
       SystemInvariantVDM[sys']                          and
       RelCalc/dom[sys'.fileStore.map] = RelCalc/dom[sys.fileStore.map] - full_path
  }
}
Check_Delete_System: check Delete_System


assert Delete_Exception_StatusUnknown {
  all sys: System, full_path: Path, status: FFS_StatusUnknown {
    SystemInvariantVDM[sys]                                 and
    PathInvariantVDM[full_path]                             and
    not (isElemFileStore[full_path, sys.fileStore] and
         pre_FS_DeleteFileDir_System[sys,full_path]      ) and
    FS_DeleteFileDir_Exception[sys,full_path,status]
    => not status = FFS_StatusUnknown
  }
}
Check_Delete_Exception_StatusUnknown: check Delete_Exception_StatusUnknown


assert Delete_Exception_FileNotFound {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                     and
    PathInvariantVDM[full_path]                 and
    not isElemFileStore[full_path,sys.fileStore]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileNotFound]
  }
}
Check_Delete_Exception_FileNotFound: check Delete_Exception_FileNotFound
```

```
assert Delete_Exception_FileStillOpen {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                    and
    PathInvariantVDM[full_path]                and
    isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorFileStillOpen]
  }
}
Check_Delete_Exception_FileStillOpen: check Delete_Exception_FileStillOpen


assert Delete_Exception_DirectoryNonEmpty {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]                               and
    PathInvariantVDM[full_path]                           and
    isDirectory[(full_path.(sys.fileStore.map)).info] and
    hasSubFiles[sys.fileStore,full_path]                  and
    not isElemTablePath[full_path,sys.table]
    => FS_DeleteFileDir_Exception[sys,full_path,FS_ErrorDirectoryNonEmpty]
  }
}
Check_Delete_Exception_DirectoryNonEmpty: check Delete_Exception_DirectoryNonEmpty


assert Delete_Main_Success {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                   and
    PathInvariantVDM[full_path]               and
    full_path in RelCalc/dom[sys.fileStore.map] and
    pre_FS_DeleteFileDir_System[sys,full_path]  and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys'] and
       status = FFS_StatusSuccess
  }
}
Check_Delete_Main_Success: check Delete_Main_Success


assert Delete_Main_Error {
  all sys,sys': System, full_path: Path, status: FFS_Status {
    SystemInvariantVDM[sys]                          and
    PathInvariantVDM[full_path]                      and
    not (full_path in RelCalc/dom[sys.fileStore.map] and
         pre_FS_DeleteFileDir_System[sys,full_path]) and
    FS_DeleteFileDir_Main[sys,sys',full_path,status]
    => SystemInvariantVDM[sys']       and
       status not = FFS_StatusSuccess
  }
}
Check_Delete_Main_Error: check Delete_Main_Error


assert checkOpenMode_CreateNew {
  all sys: System, path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path] and
```

```
    isElemFileStore[path,sys.fileStore]
    => not checkOpenMode[sys,path,FS_CreateNew]
  }
}
Check_checkOpenMode_CreateNew: check checkOpenMode_CreateNew



assert checkOpenMode_CreateAlways {
  all sys: System, path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path] and
    isElemTablePath[path,sys.table]
    => not checkOpenMode[sys,path,FS_CreateAlways]
  }
}
Check_checkOpenMode_CreateAlways: check checkOpenMode_CreateAlways


assert checkOpenMode_OpenRead {
  all sys: System, path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path] and
    not isElemFileStore[path,sys.fileStore]
    => not checkOpenMode[sys,path,FS_OpenRead]
  }
}
Check_checkOpenMode_OpenRead: check checkOpenMode_OpenRead


assert checkOpenMode_OpenWrite {
  all sys: System, path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path] and
    not isElemFileStore[path,sys.fileStore]
    => not checkOpenMode[sys,path,FS_OpenWrite]
  }
}
Check_checkOpenMode_CreateWrite: check checkOpenMode_OpenWrite


assert Open_Table {
  all table,table': OpenFilesTable, full_path: Path, omode: FS_OpenMode,
    handle: OptionalFileHandle, ft: FileType {
    OpenFilesTableInvariantVDM[table]                          and
    PathInvariantVDM[full_path]                                and
    FS_OpenFileDir_Table[table,table',full_path,omode,ft,handle]
    => ft in RegularFile
       =>   full_path = handle.(table'.map).path                          and
            RelCalc/dom[table'.map] = RelCalc/dom[table.map] + handle     and
            OpenFilesTableInvariantVDM[table']                            and
            fs_open2access_mode_map[omode,handle.(table'.map).accessMode]
       else table'.map = table.map
  }
}
Check_Open_Table: check Open_Table
                  for 7 but 0 System,
                            0 FileStore,
```

```
                             2 OpenFilesTable


assert Open_Table_Directories {
  all table,table': OpenFilesTable, full_path: Path,
      omode: FS_OpenMode, handle: OptionalFileHandle, ft: FileType {
    ft = Directory                                               and
    OpenFilesTableInvariantVDM[table]                            and
    PathInvariantVDM[full_path]                                  and
    FS_OpenFileDir_Table[table,table',full_path,omode,ft,handle]
    => full_path not in handle.(table'.map).path
  }
}
Check_Open_Table_Directories: check Open_Table_Directories
                              for 7 but 0 System,
                                        0 FileStore,
                                        2 OpenFilesTable


assert Open_FileStore {
  all fs,fs': FileStore, full_path: Path, attr: Attributes {
   FileStoreInvariantVDM[fs]                           and
   PathInvariantVDM[full_path]                         and
   pre_FS_OpenFileDir_FileStore[fs,full_path,attr] and
   FS_OpenFileDir_FileStore[fs,fs',full_path,attr]
   => FileStoreInvariantVDM[fs']            and
      full_path in RelCalc/dom[fs'.map]   and
      (not isElemFileStore[full_path,fs]
       => full_path.(fs'.map).info.attributes = attr)
  }
}
Check_Open_FileStore: check Open_FileStore
                      for 7 but 0 System,
                                0 OpenFilesTable


assert teste {
  all fs,fs': FileStore, full_path: Path, attr: Attributes {
   FileStoreInvariantVDM[fs]                           and
   PathInvariantVDM[full_path]                         and
   full_path.dirName not in RelCalc/dom[fs.map] and
   full_path not = Root and
   pre_FS_OpenFileDir_FileStore[fs,full_path,attr] and
   FS_OpenFileDir_FileStore[fs,fs',full_path,attr]
    => FileStoreInvariantVDM[fs']
  }
}
check teste


assert Open_System {
  all sys,sys': System, full_path: Path, attr: Attributes,
      omode: FS_OpenMode, amode: FS_AccessMode, handle: OptionalFileHandle {
    fs_open2access_mode_map[omode,amode]                       and
    SystemInvariantVDM[sys]                                    and
    PathInvariantVDM[full_path]                                and
    pre_FS_OpenFileDir_System[sys,full_path,attr,omode]        and
    FS_OpenFileDir_System[sys,sys',full_path,attr,omode,handle]
```

```
    => SystemInvariantVDM[sys']                                          and
       full_path in RelCalc/dom[sys'.fileStore.map]                      and
       (not isElemFileStore[full_path,sys.fileStore]
        => full_path.(sys'.fileStore.map).info.attributes = attr)        and
       (isDirectory[full_path.(sys'.fileStore.map).info]
        =>   (full_path not in FS_FileHandle.(sys'.table.map).path and
               handle in NilFileHandle)
        else (full_path = handle.(sys'.table.map).path) and
               handle.(sys'.table.map).accessMode = amode)
  }
}
Check_Open_System: check Open_System
                   for 7 but 2 System


assert Open_System_Directories {
  all sys,sys': System, full_path: Path, attr: Attributes,
      omode: FS_OpenMode, amode: FS_AccessMode, handle: OptionalFileHandle {
    full_path in RelCalc/dom[sys.fileStore.map]                  and
    isDirectory[full_path.(sys.fileStore.map).info]              and
    fs_open2access_mode_map[omode,amode]                         and
    SystemInvariantVDM[sys]                                      and
    PathInvariantVDM[full_path]                                  and
    pre_FS_OpenFileDir_System[sys,full_path,attr,omode]          and
    FS_OpenFileDir_System[sys,sys',full_path,attr,omode,handle]
    => full_path not in FS_FileHandle.(sys'.table.map).path
  }
}
Check_Open_System_Directories: check Open_System_Directories
                               for 7 but 2 System


assert Open_Exception_StatusUnknown {
  all sys: System, path: Path, omode: FS_OpenMode,
      attr: Attributes, status: FFS_Status {
    SystemInvariantVDM[sys]                          and
    PathInvariantVDM[path]                           and
    not (pre_FS_OpenFileDir_System[sys,path,attr,omode] and
         checkOpenMode[sys,path,omode])  and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => not status in FFS_StatusUnknown
  }
}
Check_Open_Exception_StatusUnknown: check Open_Exception_StatusUnknown
                                    for 2 but 1 System,
                                              1 FileStore,
                                              1 OpenFilesTable


assert Open_Exception_FileAlreadyExists {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]              and
    PathInvariantVDM[path]               and
    isElemFileStore[path,sys.fileStore]  and
    isCreateNew[omode]                   and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileAlreadyExists
  }
```

```
}
Check_Open_Exception_FileAlreadyExists: check Open_Exception_FileAlreadyExists
                                        for 7


assert Open_Exception_FileStillOpen {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]          and
    PathInvariantVDM[path]           and
    isElemTablePath[path,sys.table] and
    isCreateAlways[omode]            and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileStillOpen
  }
}
Check_Open_Exception_FileStillOpen: check Open_Exception_FileStillOpen
                                    for 7 but 1 System


assert Open_Exception_FileNotFound {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]                  and
    PathInvariantVDM[path]                   and
    not isElemFileStore[path,sys.fileStore] and
    isOpen[omode]                            and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorFileNotFound
  }
}
Check_Open_Exception_FileNotFound: check Open_Exception_FileNotFound
                                   for 7


assert Open_Exception_InvalidParameter {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]                          and
    PathInvariantVDM[path]                           and
    isOpen[omode]                                    and
    isElemFileStore[path, sys.fileStore]             and
    not isRegularFile[sys.fileStore.map[path].info] and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FFS_StatusInvalidParameter
  }
}
Check_Open_Exception_InvalidParameter: check Open_Exception_InvalidParameter
                                       for 7


assert Open_Exception_InvalidPath {
  all sys: System, path: Path, omode: FS_OpenMode, status: FFS_Status {
    SystemInvariantVDM[sys]                  and
    PathInvariantVDM[path]                   and
    (not path in Root and
     (not isElemFileStore[path.dirName,sys.fileStore] or
      not isDirectory[(sys.fileStore.map[path.dirName]).info])) and
    FS_OpenFileDir_Exception[sys,path,omode,status]
    => status = FS_ErrorInvalidPath
  }
```

```
}
Check_Open_Exception_InvalidPath: check Open_Exception_InvalidPath
                                  for 7


assert Open_Main_Success {
  all sys,sys': System, full_path: Path,
      attr: Attributes, omode: FS_OpenMode,
      handle: OptionalFileHandle, status: FFS_Status {
    PathInvariantVDM[full_path]                        and
    checkOpenMode[sys,full_path,omode]                 and
    pre_FS_OpenFileDir_System[sys,full_path,attr,omode] and
    FS_OpenFileDir_Main[sys,sys',full_path,attr,omode,handle,status]
    => SystemInvariantVDM[sys'] and
       status = FFS_StatusSuccess
  }
}
Check_Open_Main_Success: check Open_Main_Success
                         for 7


assert Open_Main_Error {
  all sys,sys': System, full_path: Path,
      attr: Attributes, omode: FS_OpenMode,
      handle: NilFileHandle, status: FFS_Status {
    SystemInvariantVDM[sys]            and
    PathInvariantVDM[full_path]        and
    not(pre_FS_OpenFileDir_System[sys,full_path,attr,omode] and
        checkOpenMode[sys,full_path,omode]) and
    FS_OpenFileDir_Main[sys,sys',full_path,attr,omode,handle,status]
    => SystemInvariantVDM[sys'] and
       status not = FFS_StatusSuccess
  }
}
Check_Open_Main_Error: check Open_Main_Error
                       for 7
```

B.4.3 Proof Obligations

```
module ProofObligation
open RelCalc
open FlashFileSystemCore
open FileSystemLayerBase
open FileSystemLayerOperations


/*
Integrity property #1 :
In type FileSystemLayerBase System, file: FileSystemLayerBase.vpp l. 26 c. 32: map
    application
-------------------------------------------------------------------------------------
(forall sys : System &
(forall ofi in set rng (sys.table) &
isElemFileStore(ofi.path, sys.fileStore) =>
 ofi.path in set dom (sys.fileStore)))
*/
```

```
assert po1 {
  all sys: System {
    SystemInvariantVDM[sys]
    => all ofi: FS_OpenFileInfo {
         FS_OpenFileInfoInvariantVDM[ofi]
         => isElemFileStore[ofi.path, sys.fileStore]
            => ofi.path in RelCalc/dom[sys.fileStore.map]
       }
  }
}
CheckP01: check po1 for 7


/*
Integrity property #1 :
In function FileSystemLayerBase newFileHandle, file: FileSystemLayerBase.vpp l. 323 c.
    38: function application from max
-------------------------------------------------------------------------------------
(forall handles : set of FS_FileHandle &
not (card (handles) = 0) =>
 FileSystemLayerBase`pre_max(handles))
 */
 assert po2 {
   all handles: set FS_FileHandle {
     not (#handles = 0) => #handles > 0
   }
 }
 CheckP02: check po2 for 7


/*
 Integrity property #2 :
In function FileSystemLayerBase newFileHandle, file: FileSystemLayerBase.vpp l. 321 c.
    1: post condition
-------------------------------------------------------------------------------------
(forall handles : set of FS_FileHandle &
FileSystemLayerBase`post_newFileHandle(handles, (if card (handles) = 0 then
1
else
max(handles) + 1)))
*/
--assert po3 {
--}
--CheckP03: check po3 for 7


/*
Integrity property #1 :
In function FileSystemLayerBase dirName, file: FileSystemLayerBase.vpp l. 146 c. 46:
    subtype
-------------------------------------------------------------------------------------
(forall full_path : Path &
not (full_path = (<Root>)) =>
 not ((exists [xx_2] : Path &
full_path = [xx_2])) =>
 is_(full_path,seq of FileName))
*/
assert po4 {
```

```
  all full_path: Path {
    PathInvariantVDM[full_path]
    => not (full_path in Root)
       => not(some xx_2: Path {
                PathInvariantVDM[xx_2]
                => full_path = xx_2
              })
          => full_path in FileNames
  }
}
CheckPO4: check po4 for 7


/*
Integrity property #2 :
In function FileSystemLayerBase dirName, file: FileSystemLayerBase.vpp l. 146 c. 17:
    subtype
-------------------------------------------------------------------------------------
(forall full_path : Path &
not (full_path = (<Root>)) =>
 not ((exists [xx_2] : Path &
full_path = [xx_2])) =>
 (forall i in set inds (full_path) &
i < len (full_path) =>
 is_(full_path,seq of FileName)))
*/
--assert po5 {
--}
--CheckPO5: check po5 for 7


/*
Integrity property #3 :
In function FileSystemLayerBase dirName, file: FileSystemLayerBase.vpp l. 146 c. 66:
    subtype
-------------------------------------------------------------------------------------
(forall full_path : Path &
not (full_path = (<Root>)) =>
 not ((exists [xx_2] : Path &
full_path = [xx_2])) =>
 (forall i in set inds (full_path) &
is_(full_path,seq of FileName)))
*/
--assert po6 {
--}
--CheckPO6: check po6 for 7


/*
 *
Integrity property #1 :
In function FileSystemLayerBase max, file: FileSystemLayerBase.vpp l. 333 c. 14: non
    emptiness of binding
-------------------------------------------------------------------------------------
(forall s : set of nat &
card (s) > 0 =>
 (exists {result} : set of nat &
{result} = {x | x in set s & (forall y in set s &
```

```
x >= y)}))
*/
--assert po7 {
--}
--CheckPO6: check po6 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 63 c. 9: invariants from FileStore
------------------------------------------------------------------------------------------
(forall fileStore : FileStore, paths : set of Path &
(forall path in set dom (fileStore) &
dirName(path) in set paths =>
 path in set paths) =>
 FileSystemLayerOperations`inv_FileStore(paths <-: fileStore))
*/
assert po8 {
  all fs,fs': FileStore, paths: set Path {
    FileStoreInvariantVDM[fs] and
    PathInvariantVDM[paths]
    => ((((Path - paths)->Path) & iden).(fs.map) in dirName.((Path - paths)->File))
       => fs'.map = fs.map - (paths->paths.(fs.map))
          => FileStoreInvariantVDM[fs']
  }
}
CheckPO8: check po8 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_Exception, file:
    FileSystemLayerOperations.vpp l. 87 c. 35: map application
------------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
not (not (isElemFileStore(full_path, sys.fileStore))) =>
 not (isElemTablePath(full_path, sys.table)) =>
 full_path in set dom (sys.fileStore))
*/
assert po9 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => not (not (isElemFileStore[full_path, sys.fileStore]))
       => not (isElemTablePath[full_path, sys.table])
          => full_path in RelCalc/dom[sys.fileStore.map]
  }
}
CheckPO9: check po9 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 213 c. 3: invariants from FileStore
------------------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
```

```
(full_path = <Root> and
 attributes.fileType = <Directory>) or
 ((let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) and
 isDirectory(fileStore(parent).info))) =>
 FileSystemLayerOperations`inv_FileStore((if not (isElemFileStore(full_path, fileStore))
      then
(let content = emptyFileContents(attributes.fileType),
     newFile = mk_File(newFileDirInfo(attributes),content)
in
    fileStore munion {full_path |-> newFile})
else
fileStore)))
*/
assert po10 {
  all fileStore, fileStore': FileStore,
      full_path: Path, attr: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => (full_path in Root and attr.fileType in Directory) or
       (let parent = full_path.dirName |
        isElemFileStore[parent, fileStore] and
        isDirectory[fileStore.map[parent].info])
       => (not isElemFileStore[full_path, fileStore]
           =>   (one newFile: File {
                   fileStore'.map = fileStore.map + (full_path -> newFile)          and
                   newFile.info.attributes = attr                                  and
                   (attr.fileType in Directory => newFile.contents in NilFileContents)
                       and
                   (attr.fileType in RegularFile => newFile.contents in FileContents)
                       and
                   not isElemFileStore[newFile,fileStore']
                })
               => FileStoreInvariantVDM[fileStore']
           else FileStoreInvariantVDM[fileStore])
  }
}
CheckPO10: check po10 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 216 c. 18: compatible maps
--------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
(full_path = <Root> and
 attributes.fileType = <Directory>) or
 ((let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) and
 isDirectory(fileStore(parent).info))) =>
 not (isElemFileStore(full_path, fileStore)) =>
 (let content = emptyFileContents(attributes.fileType),
      newFile = mk_File(newFileDirInfo(attributes),content)
in
    (forall id_9 in set dom (fileStore), id_10 in set dom ({full_path |-> newFile}) &
```

```
id_9 = id_10 =>
 fileStore(id_9) = {full_path |-> newFile}(id_10))))
*/
assert po11 {
  all fileStore: FileStore,
      full_path: Path, attr: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => (full_path in Root and attr.fileType in Directory) or
       (let parent = full_path.dirName |
        isElemFileStore[parent, fileStore] and
        isDirectory[fileStore.map[parent].info])
       => not isElemFileStore[full_path, fileStore]
          =>  (some newFile: File {
                  newFile.info.attributes = attr                                          and
                  (attr.fileType in Directory => newFile.contents in NilFileContents) and
                  (attr.fileType in RegularFile => newFile.contents in FileContents)
                  => all id_9,id_10: Path {
                         PathInvariantVDM[id_9]  and
                         PathInvariantVDM[id_10]
                         => id_9  in RelCalc/dom[fileStore.map] and
                            id_10 in RelCalc/dom[(full_path->newFile)]
                            => id_9 = id_10
                               => fileStore.map[id_9] = (full_path->newFile)[id_10]
                     }
              })
  }
}
CheckPO11: check po11 for 7


/*
Integrity property #3 :
In function FileSystemLayerOperations FS_OpenFileDir_FileStore, file:
    FileSystemLayerOperations.vpp l. 220 c. 65: map application
------------------------------------------------------------------------------------
(forall fileStore : FileStore, full_path : Path, attributes : Attributes &
not ((full_path = <Root> and
 attributes.fileType = <Directory>)) =>
 (let parent = dirName(full_path)
in
    isElemFileStore(parent, fileStore) =>
 parent in set dom (fileStore)))
*/
assert po12 {
  all fileStore: FileStore, full_path: Path, attributes: Attributes {
    FileStoreInvariantVDM[fileStore] and
    PathInvariantVDM[full_path]
    => not (full_path in Root and attributes.fileType in Directory)
       => let parent = full_path.dirName {
              isElemFileStore[parent, fileStore]
              => parent in RelCalc/dom[fileStore.map]
          }
  }
}
CheckPO12: check po12 for 7
```

```
/*
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_Exception, file:
    FileSystemLayerOperations.vpp l. 260 c. 41: map application
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, omode : FS_OpenMode &
not (isCreateNew(omode) and
 isElemFileStore(full_path, sys.fileStore)) =>
 not (isCreateAlways(omode) and
 isElemTablePath(full_path, sys.table)) =>
 not (isOpen(omode) and
 not (isElemFileStore(full_path, sys.fileStore))) =>
 isOpen(omode) and
 isElemFileStore(full_path, sys.fileStore) =>
 full_path in set dom (sys.fileStore))
*/
assert po13 {
  all sys: System, full_path: Path, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => not (isCreateNew[omode] and isElemFileStore[full_path, sys.fileStore])
       => not (isCreateAlways[omode] and isElemTablePath[full_path, sys.table])
          => not (isOpen[omode] and not isElemFileStore[full_path, sys.fileStore])
             => isOpen[omode] and isElemFileStore[full_path, sys.fileStore]
                => full_path in RelCalc/dom[sys.fileStore.map]
  }
}
CheckPO13: check po13 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_Exception, file:
    FileSystemLayerOperations.vpp l. 264 c. 61: map application
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, omode : FS_OpenMode &
not (isCreateNew(omode) and
 isElemFileStore(full_path, sys.fileStore)) =>
 not (isCreateAlways(omode) and
 isElemTablePath(full_path, sys.table)) =>
 not (isOpen(omode) and
 not (isElemFileStore(full_path, sys.fileStore))) =>
 not (isOpen(omode) and
 isElemFileStore(full_path, sys.fileStore) and
 not (isRegularFile(sys.fileStore(full_path).info))) =>
 not (full_path <> <Root> and
 not (isElemFileStore(dirName(full_path), sys.fileStore))) =>
 full_path <> <Root> =>
 dirName(full_path) in set dom (sys.fileStore))
*/
assert po14 {
  all sys: System, full_path: Path, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => not (isCreateNew[omode] and isElemFileStore[full_path, sys.fileStore])
       => not (isCreateAlways[omode] and isElemTablePath[full_path, sys.table])
          => not (isOpen[omode] and not isElemFileStore[full_path, sys.fileStore])
             => not (isOpen[omode] and isElemFileStore[full_path, sys.fileStore]
```

```
                                 and not    isRegularFile[full_path.(sys.fileStore.map).
                                     info])
                 => not (full_path in Root and not isElemFileStore[full_path.dirName, sys
                    .fileStore])
                    => full_path not in Root
                       => full_path.dirName in RelCalc/dom[sys.fileStore.map]
  }
}
CheckPO14: check po14 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 43 c. 3: invariants from System
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations`inv_System(mu(sys,fileStore|->FS_DeleteFileDir_FileStore(sys.
     fileStore, {full_path}))))
*/
assert po15 {
  all sys,sys': System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FS_DeleteFileDir_FileStore[sys.fileStore,sys'.fileStore,{full_path}] and
          sys'.table = sys.table
          => SystemInvariantVDM[sys']
  }
}
CheckPO15: check po15 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 43 c. 51: invariants from FileSystemLayerBase`
    FileStore
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerBase`inv_FileStore(FS_DeleteFileDir_FileStore(sys.fileStore, {full_path
     })))
*/
assert po16 {
  all sys: System, full_path: Path, fs': FileStore {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FS_DeleteFileDir_FileStore[sys.fileStore,fs',{full_path}]
          => FileStoreInvariantVDM[fs']
```

```
  }
}
CheckPO16: check po16 for 7


/*
Integrity property #3 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 43 c. 55: invariants from FileStore
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations`inv_FileStore(sys.fileStore))
*/
assert po17 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => FileStoreInvariantVDM[sys.fileStore]
  }
}
CheckPO17: check po17 for 7


/*
Integrity property #4 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 43 c. 51: function application from
    FS_DeleteFileDir_FileStore
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) and
 pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}) =>
 FileSystemLayerOperations`pre_FS_DeleteFileDir_FileStore(sys.fileStore, {full_path}))
*/
assert po18 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path          and
       pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
       => pre_FS_DeleteFileDir_FileStore[sys.fileStore,{full_path}]
  }
}
CheckPO18: check po18 for 7


/*
Integrity property #5 :
In function FileSystemLayerOperations FS_DeleteFileDir_System, file:
    FileSystemLayerOperations.vpp l. 45 c. 39: invariants from FileStore
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
```

```
((forall ofi in set rng (sys.table) &
ofi.path <> full_path)) =>
 FileSystemLayerOperations`inv_FileStore(sys.fileStore))
*/
assert po19 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[full_path]
    => full_path not in RelCalc/rng[sys.table.map].path
       => FileStoreInvariantVDM[sys.fileStore]
 }
}
CheckPO19: check po19 for 7



/*
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 164 c. 3e: invariants from System
----------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 FileSystemLayerOperations`inv_System((let fileStore' = FS_OpenFileDir_FileStore(sys.
     fileStore, full_path, attr), mk_(table,handle) = FS_OpenFileDir_Table(sys.table,
     full_path, omode, attr.fileType), offset = getOpenOffset(fileStore'(full_path),
     omode), table' = table ++ (if handle <> nil and
 isElemTableHandle(handle, table) then
{handle |-> mu(table(handle),fileOffset|->offset)}
else
{|->})
in
    mk_(mk_System(table',fileStore'),handle)).#1))
*/
assert po20 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle, offset: FileContents,
              table',t': OpenFilesTable, ofi: FS_OpenFileInfo {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
                and
            getOpenOffset[fileStore'.map[full_path], omode, offset] and
            (handle not in NilFileHandle and isElemTableHandle[handle, t]
             =>   (ofi.accessMode = t.map[handle].accessMode and
                   ofi.path = t.map[handle].path             and
                   ofi.fileOffset = offset
                   => t'.map = (handle->ofi))
             else no t'.map)
            table'.map = (t.map - (RelCalc/dom[t'.map]->FS_OpenFileInfo)) + t'.map
            => all sys': System {
                 sys'.table = table'         and
                 sys'.fileStore = fileStore'
                 => SystemInvariantVDM[sys']
               }
          }
```

```
  }
}
CheckPO20: check po20 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 164 c. 55: invariants from FileStore
------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 FileSystemLayerOperations`inv_FileStore(sys.fileStore))
*/
assert po21 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => FileStoreInvariantVDM[sys.fileStore]
  }
}
CheckPO21: check po21 for 7


/*
Integrity property #3 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 164 c. 51: function application from
    FS_OpenFileDir_FileStore
------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 FileSystemLayerOperations`pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr))
*/
assert po22 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
  }
}
CheckPO22: check po22 for 7


/*
Integrity property #4 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 166 c. 48: invariants from File
------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType)
in
    FileSystemLayerOperations`inv_File(fileStore'(full_path))))
```

```
*/
assert po23 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => FileInvariantVDM[fileStore'.map[full_path]]
          }
  }
}
CheckPO23: check po23 for 7


/*
Integrity property #5 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 166 c. 48: map application
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType)
in
    full_path in set dom (fileStore')))
*/
assert po24 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => full_path in RelCalc/dom[fileStore'.map]
          }
  }
}
CheckPO24: check po24 for 7


/*
Integrity property #6 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 167 c. 74: subtype
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr), mk_(table,
     handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.fileType), offset
     = getOpenOffset(fileStore'(full_path), omode)
in
    handle <> nil =>
```

```
 is_FileSystemLayerBase`FS_FileHandle(FileSystemLayerOperations`handle)))
*/
--assert po25 {
--}
--CheckPO25: check po25 for 7


/*
Integrity property #7 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 168 c. 83: subtype
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
    offset = getOpenOffset(fileStore'(full_path), omode)
in
    handle <> nil and
 isElemTableHandle(handle, table) =>
 is_nat1(offset)))
*/
assert po26 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => handle not in NilFileHandle
               => isElemTableHandle[handle, t]
          }
  }
}
CheckPO26: check po26 for 7


/*
Integrity property #8 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 168 c. 59: subtype
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr), mk_(table,
     handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.fileType), offset
     = getOpenOffset(fileStore'(full_path), omode)
in
    handle <> nil and
 isElemTableHandle(handle, table) =>
 is_FileSystemLayerBase`FS_FileHandle(FileSystemLayerOperations`handle)))
*/
assert po27 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
```

```
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => handle not in NilFileHandle and isElemTableHandle[handle, t]
               => handle in FS_FileHandle
          }
  }
}
CheckPO27: check po27 for 7


/*
Integrity property #9 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 168 c. 58: map application
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset = getOpenOffset(fileStore'(full_path), omode)
in
    handle <> nil and
 isElemTableHandle(handle, table) =>
 handle in set dom (table)))
*/
assert po28 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
            => handle not in NilFileHandle and isElemTableHandle[handle, t]
               => handle in RelCalc/dom[t.map]
          }
  }
}
CheckPO28: check po28 for 7


/*
Integrity property #10 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 170 c. 17: subtype
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr), mk_(table,
     handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.fileType), offset
     = getOpenOffset(fileStore'(full_path), omode), table' = table ++ (if handle <> nil
     and
```

```
 isElemTableHandle(handle, table) then
{handle |-> mu(table(handle),fileOffset|->offset)}
else
{|->})
in
    is_FileSystemLayerBase`OpenFilesTable(FileSystemLayerOperations`table')))
*/
assert po29 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]      and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle, offset: FileContents,
              table',t': OpenFilesTable, ofi: FS_OpenFileInfo {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
                and
            getOpenOffset[fileStore'.map[full_path], omode, offset] and
            (handle not in NilFileHandle and isElemTableHandle[handle, t]
             =>    (ofi.accessMode = t.map[handle].accessMode and
                    ofi.path = t.map[handle].path             and
                    ofi.fileOffset = offset
                    => t'.map = (handle->ofi))
             else no t'.map)
            table'.map = (t.map - (RelCalc/dom[t'.map]->FS_OpenFileInfo)) + t'.map
            => OpenFilesTableInvariantVDM[table']
          }
  }
}
CheckPO29: check po29 for 7


/*
Integrity property #11 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 170 c. 25: invariants from FileSystemLayerBase`
    FileStore
------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr) =>
 (let fileStore' = FS_OpenFileDir_FileStore(sys.fileStore, full_path, attr),
      mk_(table,handle) = FS_OpenFileDir_Table(sys.table, full_path, omode, attr.
          fileType),
      offset = getOpenOffset(fileStore'(full_path), omode),
    table' = table ++ (if handle <> nil and isElemTableHandle(handle, table)
                       then {handle |-> mu(table(handle),fileOffset|->offset)}
                         else {|->})
  in
    FileSystemLayerBase`inv_FileStore(fileStore')))
*/
assert po30 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]      and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_FileStore[sys.fileStore, full_path, attr]
       => all fileStore': FileStore, t: OpenFilesTable,
              handle: OptionalFileHandle, offset: FileContents,
```

```
              table',t': OpenFilesTable, ofi: FS_OpenFileInfo {
            FS_OpenFileDir_FileStore[sys.fileStore, fileStore', full_path, attr] and
            FS_OpenFileDir_Table[sys.table, t, full_path, omode, attr.fileType, handle]
                and
            getOpenOffset[fileStore'.map[full_path], omode, offset] and
            (handle not in NilFileHandle and isElemTableHandle[handle, t]
             =>   (ofi.accessMode = t.map[handle].accessMode and
                   ofi.path = t.map[handle].path             and
                   ofi.fileOffset = offset
                   => t'.map = (handle->ofi))
             else no t'.map)
            table'.map = (t.map - (RelCalc/dom[t'.map]->FS_OpenFileInfo)) + t'.map
            => FileStoreInvariantVDM[fileStore']
          }
  }
}
CheckPO30: check po30 for 7


/*
Integrity property #12 :
In function FileSystemLayerOperations FS_OpenFileDir_System, file:
    FileSystemLayerOperations.vpp l. 171 c. 37: invariants from FileStore
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attr : Attributes, omode : FS_OpenMode &
FileSystemLayerOperations`inv_FileStore(sys.fileStore))
*/
assert po31 {
  all sys: System, full_path: Path, attr: Attributes, omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => FileStoreInvariantVDM[sys.fileStore]
  }
}
CheckPO31: check po31 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_DeleteFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 25 c. 35: function application from
    FS_DeleteFileDir_System
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path &
full_path in set dom (sys.fileStore) and
 pre_FS_DeleteFileDir_System(sys, full_path) =>
 FileSystemLayerOperations`pre_FS_DeleteFileDir_System(sys, full_path))
*/
assert po32 {
  all sys: System, full_path: Path {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => full_path in RelCalc/dom[sys.fileStore.map] and
       pre_FS_DeleteFileDir_System[sys, full_path]
       => pre_FS_DeleteFileDir_System[sys, full_path]
  }
}
CheckPO32: check po32 for 7
```

```
/*
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_Table, file:
    FileSystemLayerOperations.vpp l. 235 c. 44: map application
---------------------------------------------------------------------------------------------
(forall table : OpenFilesTable, full_path : Path, omode : FS_OpenMode, fileType :
    FileType &
not (fileType = <Directory>) =>
 omode in set dom (fs_open2access_mode_map))
*/
assert po33 {
  all omode: FS_OpenMode, fileType: FileType {
    not (fileType in Directory)
    => some amode: FS_AccessMode {
         fs_open2access_mode_map[omode,amode]
       }
  }
}
CheckPO33: check po33 for 7
// removed table  and full_path because they are not used


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_Table, file:
    FileSystemLayerOperations.vpp l. 238 c. 18: compatible maps
---------------------------------------------------------------------------------------------
(forall table : OpenFilesTable, full_path : Path, omode : FS_OpenMode, fileType :
    FileType &
not (fileType = <Directory>) =>
 (let amode = fs_open2access_mode_map(omode),
      ofi = mk_FS_OpenFileInfo(1,amode,full_path),
    handle = newFileHandle(dom (table))
  in
    (forall id_11 in set dom (table), id_12 in set dom ({handle |-> ofi}) &
       id_11 = id_12 => table(id_11) = {handle |-> ofi}(id_12))))
*/
assert po34 {
  all table: OpenFilesTable, full_path: Path, omode: FS_OpenMode, fileType: FileType {
    not (fileType in Directory)
    => all amode: FS_AccessMode, ofi: FS_OpenFileInfo,
           handle: FS_FileHandle {
         FS_OpenFileInfoInvariantVDM[ofi]
         => fs_open2access_mode_map[omode,amode] and
            ofi.accessMode = amode               and
            ofi.path = full_path                 and
            handle not in RelCalc/dom[table.map]
            => all id_11, id_12: FS_FileHandle {
                 id_11 in RelCalc/dom[table.map]     and
                 id_12 in RelCalc/dom[(handle->ofi)]
                 => id_11 = id_12 => table.map[id_11] = (handle->ofi)[id_12]
               }
       }
  }
}
CheckPO34: check po34 for 7
```

```
/*
Integrity property #1 :
In function FileSystemLayerOperations FS_OpenFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 106 c. 32: invariants from FileStore
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attributes : Attributes, omode : FS_OpenMode &
   pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
   checkOpenMode(sys, full_path, omode) =>
     FileSystemLayerOperations`inv_FileStore(sys.fileStore))
*/
assert po35 {
  all sys: System, full_path: Path, attr: Attributes,
      omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_System[sys, full_path, attr, omode] and
       checkOpenMode[sys, full_path, omode]
       => FileStoreInvariantVDM[sys.fileStore]
  }
}
CheckPO35: check po35 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations FS_OpenFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 111 c. 52: function application from
    FS_OpenFileDir_System
-------------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attributes : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
 checkOpenMode(sys, full_path, omode) =>
 mustDeleteFirst(sys.fileStore, full_path, omode) and
 (full_path = <Root> =>
 attributes.fileType = <Directory>) =>
 (let mk_(sys',status) = FS_DeleteFileDir_Main(sys, full_path)
in
    not (status <> <FFS_StatusSuccess>) =>
 FileSystemLayerOperations`pre_FS_OpenFileDir_System(sys', full_path, attributes, omode)
     ))
*/
assert po36 {
  all sys: System, full_path: Path, attributes: Attributes,
      omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_System[sys, full_path, attributes, omode] and
       checkOpenMode[sys, full_path, omode]
       => mustDeleteFirst[sys.fileStore, full_path, omode] and
          (full_path in Root => attributes.fileType in Directory)
          => all sys': System, status: FFS_Status {
               status not in FFS_StatusSuccess
               => pre_FS_OpenFileDir_System[sys', full_path, attributes, omode]
             }
  }
}
```

```
CheckPO36: check po36 for 7


/*
Integrity property #3 :
In function FileSystemLayerOperations FS_OpenFileDir_Main, file:
    FileSystemLayerOperations.vpp l. 113 c. 58: function application from
    FS_OpenFileDir_System
-------------------------------------------------------------------------------
(forall sys : System, full_path : Path, attributes : Attributes, omode : FS_OpenMode &
pre_FS_OpenFileDir_System(sys, full_path, attributes, omode) and
 checkOpenMode(sys, full_path, omode) =>
 not (mustDeleteFirst(sys.fileStore, full_path, omode) and
 (full_path = <Root> =>
 attributes.fileType = <Directory>)) =>
 FileSystemLayerOperations`pre_FS_OpenFileDir_System(sys, full_path, attributes, omode))
*/
assert po37 {
  all sys: System, full_path: Path, attributes: Attributes,
      omode: FS_OpenMode {
    SystemInvariantVDM[sys]     and
    PathInvariantVDM[full_path]
    => pre_FS_OpenFileDir_System[sys, full_path, attributes, omode] and
       checkOpenMode[sys, full_path, omode]
       => not (mustDeleteFirst[sys.fileStore, full_path, omode] and
               (full_path in Root => attributes.fileType in Directory))
          => pre_FS_OpenFileDir_System[sys, full_path, attributes, omode]
  }
}
CheckPO37: check po37 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations FS_Init_FileStore, file: FileSystemLayerOperations
    .vpp l. 392 c. 3: invariants from FileStore
-------------------------------------------------------------------------------
FileSystemLayerOperations`inv_FileStore({<Root> |-> mk_File(mk_FS_FileDirInfo(
    mk_Attributes(<Directory>)),nil)})
*/
assert po38 {
  all file: File, fs: FileStore {
    FileInvariantVDM[file] and
    PathInvariantVDM[Root]
    => file.info.attributes.fileType = Directory and
       file.contents = NilFileContents           and
       fs.map = (Root->file)
       => FileStoreInvariantVDM[fs]
  }
}
CheckPO38: check po38 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations getOpenOffset, file: FileSystemLayerOperations.vpp
     l. 185 c. 36: subtype
-------------------------------------------------------------------------------
```

```
(forall file : File, omode : FS_OpenMode &
file.contents <> nil =>
 omode = (<FS_OpenWrite>) =>
 (let (<FS_OpenWrite>) = omode
in
    is_(file.contents,seq of token)))
*/
--assert po38 {
--}
--CheckPO38: check po38 for 7


/*
Integrity property #2 :
In function FileSystemLayerOperations getOpenOffset, file: FileSystemLayerOperations.vpp
     l. 186 c. 33: subtype
-------------------------------------------------------------------------------------
(forall file : File, omode : FS_OpenMode &
file.contents <> nil =>
 not (omode = (<FS_OpenWrite>)) =>
 omode = (<FS_OpenAlways>) =>
 (let (<FS_OpenAlways>) = omode
in
    is_(file.contents,seq of token)))
*/
--assert po39 {
--}
--CheckPO39: check po39 for 7


/*
Integrity property #1 :
In function FileSystemLayerOperations checkOpenMode, file: FileSystemLayerOperations.vpp
     l. 148 c. 36: map application
-------------------------------------------------------------------------------------
(forall sys : System, path : Path, omode : FS_OpenMode &
not ((isCreateNew(omode) and
 isElemFileStore(path, sys.fileStore))) and
 not ((isCreateAlways(omode) and
 isElemTablePath(path, sys.table))) and
 not ((isOpen(omode) and
 not (isElemFileStore(path, sys.fileStore)))) =>
 (isOpen(omode) or
 omode = <FS_OpenAlways>) and
 isElemFileStore(path, sys.fileStore) =>
 path in set dom (sys.fileStore))
*/
assert po40 {
  all sys: System, path: Path, omode: FS_OpenMode {
    SystemInvariantVDM[sys] and
    PathInvariantVDM[path]
    => not (isCreateNew[omode]    and isElemFileStore[path, sys.fileStore]) and
       not (isCreateAlways[omode] and isElemTablePath[path, sys.table])     and
       not (isOpen[omode]         and not isElemFileStore[path, sys.fileStore])
       => (isOpen[omode] or omode in FS_OpenAlways) and
          isElemFileStore[path, sys.fileStore]
          => path in RelCalc/dom[sys.fileStore.map]
  }
```

```
}
CheckPO40: check po40 for 7
```