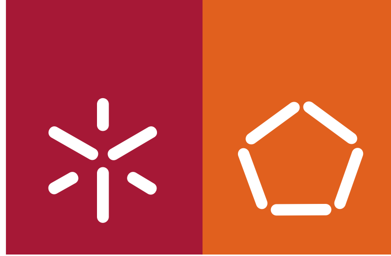
Universidade do Minho
Escola de Engenharia

Marta de Jesus Rodrigues Fernandes

Model-checking in Alloy of a Fragment of the UBIFS File System for Flash Memory

Outubro de 2010



Universidade do Minho
Escola de Engenharia

Marta de Jesus Rodrigues Fernandes

Model-checking in Alloy of a Fragment of the UBIFS File System for Flash Memory

Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Oliveira

Outubro de 2010

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/______

Assinatura: ____

Model-checking in Alloy of a Fragment of the UBIFS File System for Flash Memory

Abstract

Following the Verifiable File System (VFS) challenge proposed by *Rajeev Joshi* and *Gerard Holzmann*, the main ideia of the present thesis is to compare two abstract models of the UBIFS [1] flash file system recently included in the Linux Kernel. The thesis was developed following two approaches: on the one hand, in order to obtain comparative data and following the work of *Andreas Schierl et al*, the abstract specification developed in KIV is translated to Alloy. With such translation, we aim at evaluation

1. the differences between a more declarative language (Alloy) and a more imperative language (KIV);
2. the different verification techniques each language provides.

On the other hand, following up one of the case studies of mentioned by *José Nuno Oliveira* and the work developed by *Miguel Ferreira*, the main idea is to create an abstract model of UBIFS. The developed work based on the work of *Miguel Ferreira* focuses on building a relational model composed by four main components: the inode-based file store, the flash index, its cached copy in the RAM and the journal. The first step involves using Alloy to create the relational model and quickly verifying it due to through its capability of searching exhaustively for counter-examples to properties. The second step involves submitting the model in Alloy to the ESC-PF calculus. The main idea is building a *point-free* version of the file system based on the model in Alloy and verifying it using a different technique.

Finally, the contribution to the Grand Challenge (GC) is done by emphasizing the main differences/similarities between both models. Since most of the developed work under de GC is hard to compare (such as the verification of different file systems), the present thesis also intends to contribute with the main conclusions about the model built intentionally from the existing model in KIV.

[1] `http://www.linux-mtd.infradead.org/doc/ubifs.html`

Model-checking in Alloy of a Fragment of the UBIFS File System for Flash Memory

Resumo

No âmbito do desafio VFS proposto por *Rajeev Joshi* e *Gerard Holzmann*, o principal objectivo da presente tese consiste em desenvolver um trabalho comparativo entre dois modelos abstractos do sistema de ficheiros UBIFS para memórias *flash* incluído recentemente no *kernel* do sistema operativo Linux. O trabalho foi desenvolvido seguindo duas abordagens: por um lado, no inuito de obter dados comparativos e seguindo o trabalho de *Andreas Schierl et. al*, a especificação abstracta construída usando a ferramenta KIV é agora re-escrita em Alloy. Com a tradução de KIV para Alloy pretende-se avaliar:

1. as diferenças entre uma linguagem mais declarativa (Alloy) e uma linguagem mais imperativa (KIV);
2. as diferentes técnicas de verificação associadas a cada linguagem.

Por outro lado, dando seguimento a um dos casos de estudo mencionados por *José Nuno Oliveira* e ao trabalho desenvolvido por *Miguel Ferreira*, a ideia principal reside em criar um modelo abstracto do sistema de ficheiros UBIFS. O trabalho desenvolvido, com base no trabalho de *Miguel Ferreira*, foca-se na construção de um modelo relacional composto principalmente por quatro estruturas de armazenamento: *flash store*, *flash index*, a sua cópia na memória RAM (*RAM index*) e *journal*. O primeiro passo envolve a criação do modelo relacional com recurso à ferramenta Alloy, devido à sua capacidade de procura exaustiva de contra-exemplos para uma determinada propriedade. A capacidade de transformar Lógica relacional em Lógica booleana simplifica a sua integração com a notação *pointfree*. Dado que a notação-PF facilita a escrita de relação e propriedades, o mapeamento entre Alloy e a notação-PF torna-se assim mais simples e directo.

Por último, a contribuição para o Grande Desafio é feita através do ênfase dado às principais diferenças/semelhanças entre os dois modelos. Uma vez que grande parte do trabalho realizado no contexto do GC é de difícil comparação (como por exemplo a verificação de diferentes sistemas de ficheiros), a presente tese pretende contruibir também com as principais conclusões acerca do modelo desenvolvido que intencionalmente decalca o modelo KIV existente.

Acknowledgments

First of all, I would like to thank José Nuno Oliveira for accepting to supervise my master thesis. I would also like to thank all his help, effort and interest in this project since without his valuable and indispensable guidance, the work carried out would not have been possible.

To Miguel Ferreira the most sincere acknowledgement for his valuable help and availability, particularly in reviewing of the model in Alloy, under his co-supervision. I also thank his aid in reviewing this master thesis, further providing all the necessary documentation to its writing.

I would also like to thank to Gerhard Schellhorn and Andreas Schierl for making all KIV-related documentation available, including not only the full version of the tool but also the abstract specification of UBIFS. A special thanks to Gerhard Schellhorn for enlightening me regarding certain aspects of the KIV and the abstract specification of the file system.

Many thanks to Joana, Mário and Flávio for their contribution to polish English, a very helpful kind of support. Special thanks to Mário for all the *tips* and for his aid.

Many thanks to my friends (especially Ana) and family for the support and patience, particularly in the final phase of the work.

Last but not least, a special thanks to Rafael for being my foundation. His unconditional support gave me strength in the good and bad times to dedicate myself entirely to this thesis.

Contents

1 Introduction 1
1.1 Contextualization . . . 3
1.2 Formal Verification . . . 3
1.2.1 Model-checking . . . 5
1.2.2 Theorem proving . . . 7
1.2.3 Extended Static Checking . . . 7
1.3 Problem Description . . . 9
1.3.1 The UBIFS File System . . . 9
1.3.2 The State of the Art and Related Work . . . 10
1.4 Aims . . . 11
1.5 Document Structure . . . 11

2 Tool Background 13
2.1 KIV . . . 13
2.1.1 Specification Language . . . 14
2.1.2 Proof suport . . . 16
2.1.3 User interface . . . 17
2.2 Alloy . . . 20
2.2.1 Specification Language . . . 20
2.2.2 Alloy Analyzer . . . 23

3 Mapping between *KIV* and *Alloy* 27
3.1 Abstract Specification . . . 28
3.2 KIV Specification . . . 30
3.2.1 Data Types . . . 30
3.2.2 Axioms . . . 31
3.2.3 Operations . . . 32
3.2.4 Verification . . . 34
3.3 Alloy Specification . . . 35
3.3.1 Data Types . . . 35
3.3.2 Functions and Auxiliary Operations . . . 36
3.3.3 Invariants . . . 37

3.3.4 Operations 40
3.3.5 Verification 42

4 Concluding Remarks and Future Work 53
4.1 Concluding Remarks 53
4.1.1 Contributions 53
4.1.2 Overview of the tools 56
4.1.3 Difficulties 57
4.1.4 The verification in Alloy 58
4.2 Future Work 59
4.2.1 VDM model of UBIFS 59
4.2.2 A more abstract model of UBIFS using Alloy 60
4.2.3 Using ESC-PF notation in the UBIFS 61

A Models 67
A.1 KIV Model 67
A.1.1 Node specification 67
A.1.2 Key specification 67
A.1.3 Dentry specification 68
A.1.4 Store specification 68
A.1.5 Flash store specification 68
A.1.6 RAM and flash index specification 68
A.1.7 List-dup specification 69
A.1.8 Addresslist specification 69
A.1.9 Filesystem-base specification 69
A.1.10 Filesystem-asm specification 71
A.1.11 Replay process 73
A.1.12 File system Specification (log-cons invariant) 73
A.1.13 Log-cons invariant of the unlink operation 74
A.1.14 Datanode-cons invariant of the writepage operation 74
A.1.15 Proof tree of datanode_cons_unlink 74
A.1.16 Graphical overview of UBIFS 74
A.2 Alloy Model 77
A.2.1 Node signature 77
A.2.2 Key signature 77
A.2.3 UBIFS signature 77
A.2.4 Dentry signature 78
A.2.5 Function getinode 78
A.2.6 Constructors in Alloy 78
A.2.7 Predicate new_inodekey 78
A.2.8 Predicate valid-ino 79

A.2.9 Predicate fs_key_cons 79
A.2.10 Predicate fs-dentry-cons 79
A.2.11 Predicate fs-file-cons 79
A.2.12 Predicate fs-dir-cons 80
A.2.13 Predicate datanode-cons 80
A.2.14 Predicate nodekey-cons 80
A.2.15 Operation unlink 80
A.2.16 Operation writepage (old version) 81
A.2.17 Operation writepage (fixed version) 82
A.2.18 Operation create 83
A.2.19 Assertion nodekey_cons_unlink 84
A.2.20 Assertion danode_cons_writepage 84
A.2.21 Assertion prepost_unlink 84
A.2.22 Assertion prepost_readpage 84
A.2.23 Assertion prepost_writepage 85
A.2.24 Assertion log_cons_unlink 85
A.2.25 Fact Notnegative 85
A.2.26 Metamodel of the UBIFS File System 85

List of Figures

1.1 Model checking approach (quoted from [35]) 5
1.2 Static checkers comparison regarding error coverage and effort 8
1.3 RAM index, flash index, flash store and journal 10

2.1 Graphical user interface of KIV 18
2.2 Graph visualization of KIV 18
2.3 Proof visualization of KIV 19
2.4 Proof tree of KIV 19
2.5 Information of a proof 20
2.6 User interface of Alloy Analyzer 24
2.7 Metamodel of a directory entry generated by Alloy Analyzer 24
2.8 Instance found of the function new_inodekey 25

3.1 Instance of valid_ino produced by Alloy Analyzer 38
3.2 Instance of the operation writepage 43
3.3 Flash store and RAM index of u 43
3.4 Flash store and RAM index of u' 43
3.5 counter-example of datanode_cons_writepage 45
3.6 Flash store and RAM index of u 45
3.7 Flash store and RAM index of u' 45
3.8 Output of getinode under the counter-example of 3.3.5 46
3.9 counter-example of datanode_cons_writepage 46

Acronyms

VFS Verifiable File System i

MGS Mars Global Survivor 1

AST Automated Theorem Proving 7

ESC Extended Static Checking 7

UBIFS Unsorted Block Image File System 9

KIV Karlsruhe Interactive Verifier 13

PPL Proof Programing Language 13

PF Point-free 8

GC Grand Challenge i

VDM Vienna Development Method 11

HOL High Order Logic 11

SAT Propositional Satisfiability 6

ASM Abstract State Machine 27

JML Java Modeling Language 8

Chapter 1

Introduction

On the 7h of November 1996, *NASA's Jet Propulsion Laboratory* launches the spacecraft Mars Global Survivor (MGS) [1] to planet Mars. Having completed its primary mission by Janueary 2001 and engaged in its third extended mission phase, on the 2nd of November 2006, MGS stops to communicate with Earth and consequently fails to respond to messages and commands. Three days later, a faint signal is detected indicating that the spacecraft succumbed to battery failure. All attemps to regan contact with MGS and solve the problem fail and two months later NASA officially ends the mission. The accident was caused by a complex sequence of events from a software update made five months before, that involved the onboard computer memory (two memory addresses were incorrect) and ground commands.

Poor quality software often does not cause irreversible damage, a simple software reboot or even system reboot solves the problem in most cases. In safety-critical systems, however, this leads to far more serious consequences as seen in MGS. This kind of failures occur due to late discovery of critical flaws in the construction of software. Sometimes, a fatal inconsistency or omission is the cause, but more often, a key factor that leads to this kind of situations is that only after the beginning of the software development programmers discover the inadequacy of their designs. As the software development grows with additional fixes, its design erodes in detail. The result is a confusing, incoherent (and possibly unstable) design structure increasingly hard to fix whenever a fault occurs. It is in this context that the importance of formal methods is emphasized: to contribute to the reliability and robustness of a system.

One of the application fields of formal methods is formal verification whose main goal is to prove correctness of software. In order to minimize possible flaws in software construction, *Tony Hoare* proposed a grand challenge for computing research: the construction and application of a verifying compiler ensuring correctness of a program before running it [24]. Since correctness of software is the fundamental concern of the theory of programming, such verifying compiler could be used in the verification of structural integrity, security and substantial verification of critical parts of millions of lines of open source software. This could lead to removal of thousands of errors in widely used code and even to blockade of viruses entries (these often get into a computer system by exploring errors such as buffer overflow). The contribution of the verifying compiler to the development and

[1] `http://www.nasa.gov/mission_pages/mgs/mgs-20070413.html`

maintenance of new code is given by supporting its specification, design and test.

In order to collaborate with *Hoare* in the grand challenge, *R. Joshi* and *G. Holzmann* proposed a “mini challenge”: a nontrivial verification project that can nonetheless be completed in a short time [29]. *Joshi* and *Holzmann* believe that the ideal candidate for a mini challenge would have the following characteristics:

- to be sufficiently complex as to render traditional methods inadequate to prove its correctness,
- to be simple enough that specification, design and verification could be completed in a relatively short time,
- to be relevant enough so that successful competion would have impact beyond the verification community.

The fact that most modern file systems have a well defined interface and the data structures used in their design are very well understood makes a file system a good candidate for such a mini challenge. By contrast, ensuring file-system reliability in concurrent operation modes and unexpected power failure is an arduous task. Verification tools call for a number of requirements in this respect, namely:

- a formal behavioral specification of the file system,
- a formal elaboration of the assumptions made of the underlying hardware,
- a set of invariants, assertions and properties that must be valid about data structures and algorithms in the implementation

The first step of writing a behavioral specification of the file system is to understand its structure and management. Modern file systems are written according to the POSIX standard [2]: a set of operations (read, write, create, open, etc) is available together with the corresponding descripton. Another element to consider is the underlying hardware. The behavioral properties of a file system are different depending on the underlying hardware. Therefore the assertions that ensure correctness and reliability may be different (a file system for a spinning disk drive has different characteristics from a file system for flash memory drives, or from a file system that is distributed over a network). Peculiarities of flash memory drives are discussed in Section 1.1 together with a detailed description of the UBIFS.

As mentioned before, writing a verifiable file system requires a full knoweledge about design properties such as structure invariants, pre- and post-conditions of functions and how data is physically stored. Depending on the type of file system in question (regardless of the hardware), it is essential to understand its structure. For instance, there are a few differences between a *versioning* file system and a *journaling* file system. A *versioning* file system allows for simultaneous existence of several version of a file. Versioning can be implemented using two techniques: keeping a number of old copies of the modified file or taking periodic snapshots of data. However, due to storage limitations the number of versions of modified files is also limited [38]. A *journaling* file system is

a powerloss resilient file system where data integrity is ensured by the fact that data updates are constantly written to a serial log before being stored on disk. This log (called *journal*) is a especially allocated area of the file system that minimizes the possibilities of losing changes made to files. A detailed definition of journaling can be found in Section 1.1.

1.1 Contextualization

Nowadays, an increasing demand on flash memories is observed due to their advantages like shock resistance and absence of mechanical parts. However, a reliable data storage on top of flash devices requires thorough knowledge of their shortcomings: they cannot be overwritten, but only erased in blocks and this should be performed using the *wear levelling* technique[1].

Block erasure presents some inconveniences: its execution is very slow and the number of *erase* operations for a memory cell is limited [31]. This may lead to resorting of overwriting data. However, the flash memory content cannot be overwritten, i.e., to write new data on a currently used part of the flash memory that block should be erased first. Given that physically erasing data from flash memory is what wears out the memory blocks, erasing file system data should simply be achieved by marking the blocks for future garbage collection. Due to the impossibility of in-place changes of already written data, updates are written somewhere else on the flash memory. To handle this limitation and the deterioration of blocks of the flash memory that are often erased, the concept of *garbage collection*[2] is used to check if an entry in a erase block is valid or obsolete, by analyzing additional metadata. The combination of metadata and data is called a *node*.

The UBIFS is a journaled flash file system designed to work on top of raw flash [7]: nodes are stored in eraseblocks with the possibility of being written to, read from or erased. The main goals of UBIFS are better performance and scalability (according to flash size) achieved by write-back and maintaining indexing information on the flash media.

The complexity of UBIFS makes it a good candidate for the mini-challenge proposed by *Rajeev Joshi* and *Gerard Holzmann* [29]: building a *verifiably* reliable and secure file system.

1.2 Formal Verification

As software designs grows in size and complexity, conventional design methods become inadequate. Current methods for specification, design and test are typically empirical or informal and therefore not enough to spot subtle design flaws. Formal methods improve software (and hardware) designs by revealing ambiguity, incompleteness and inconsistency in a system. When used in the initial phases of the development process, they can find design flaws that otherwise would only be found by testing and debugging. When used in the final phases, they can aid to check correctness and equivalence of different implementations of a system. However, the high cost of using formal

[1] Definition of "wear levelling" from Wikipedia: http://en.wikipedia.org/wiki/Wear_levelling

[2] Definition of garbage collection from Wikipedia: http://en.wikipedia.org/wiki/Garbage_collection _(computer_science)

methods and the resistance to them by many developers has confined their application to high-integrity, safety-critical systems. Formal methods are usually associated to the usage of complex and difficult methods and techniques. On the other hand, *Alessandro Fantechi et al* report the success of formal methods in a Railway Signaling Manufacturer [4] by reducing total development effort by more than 50%. Other successful cases of using formal methods can be found in [22] and [37].

Formal methods are classified in several levels of usage, one of these being formal verification. The main goal of formal verification is ensuring the correctness of an intended design by writing properties of software or hardware designs using mathematical logic. This task involves formal specification of the requirement, formal modeling of the implementation and accurate rules of inference to prove the satisfiability of the implementation under the underlying specification. Formal verification has proved useful in ensuring the correctness of the following systems:

- **cryptographic protocols –** It is important to ensure security in cryptographic protocols due to the risky presence of eavesdroppers in the exchange of messages between two hosts. If a hostile eavesdropper succeeds in capturing messages or feeding false messages to honest users and the protocol is not designed to avoid both situations, then the intruder's action may result in some security failures. *Catherine Meadows* approaches this theme in [32].

- **combinational circuits –** successful design of a complex system requires verifying the correctness of the implementation according to its intended functionality. Traditional design validation resorts to its simulation, burdening the designer with the need to create multiple test with the aim of representing all possible inputs. Correctness is approached based on the analysis of the output for each input vector. The high cost of CPU-time leads to the impossibility of simulating exhaustively a design in order to ensure its correctness. Given the limitations of simulation based approach, several strategies of formal verification are becoming increasingly popular. Further information of the techniques for formal verification used in combinational circuits can be found in [27].

- **digital circuits –** nowadays, integrated circuits technology allows the development of chips with several millions of transistors. However, due to increasing the complexity of digital circuit designs, traditional functional verification (based on simulation) reached its limits. Therefore, technological alternatives to simulation, such as formal verification, are a growing burden. Formal verification provides quality, cost and time improvements for tasks constituting over 60% of the overall development efforts. Equivalence checking is one of the available approaches of formal verification. In digital circuits, in order to reduce the complexity of the final netlist, a set of optimizations are done in the design and developed code, leading to the existence of several previous netlists. In this context, equivalence checking is be used to check if the final netlist has the same behavior as previous netlists and even the original specification. Further information about the use of equivalence checking in digital circuits can be found in [14].

- **Source code –** one of several existing approaches to formal verification is verifying directly the source code of a software implementation. Functional properties of the system are defined by inserting annotations in the source code and a proof obligation generator is used to verify

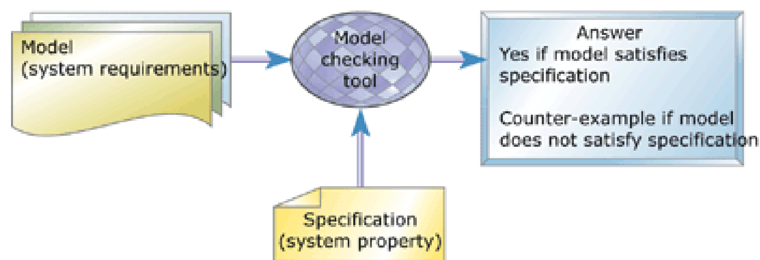


Figure 1.1: Model checking approach (quoted from [35])

if the implementation satisfies these properties. *June Andronick et al* report on a method to handle the verification of various security properties of imperative source code embedded on smart cards in [3].

1.2.1 Model-checking

Building a project involves, before starting to write lines of code, facing chronic issues of software development such as flaw requirements. Since flawed requirements generate bugs and solving them often has a high cost, it is important to find flaws beforehand. In the last decade, the computer science research community has made enormous progress in developing tools and techniques for verifying requirements and design. The most successful approach that has emerged is called model checking. When strictly used with a formal modeling language, it helps in automating the verification process fairly well. Model checking [5] is a technique of testing automatically if a model of a system meets a specification. Typically, when planning systems one thinks in software or hardware systems, and the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Hence, model checking is used to verify correctness properties of finite state systems.

Building tools to check requirements written in natural language is clearly extremely hard. It is necessary to enforce a clear, rigorous, and unambiguous formal language for stating the requirements. If the specification language for writing requirements and design has well-defined semantics, it may be appropriate to develop tools that analyze the statements written in such a language. This basic idea of using a rigorous language for writing requirements or design is now acknowledged as a foundation for system verification.

Tipically, a model-checker accepts system requirements or design (models) and some properties (specification) that the final system must satisfy. The output provided by the tool indicates if the property is satisfied or generates a counter-example if the property does not hold. By studying the counter-example, it is possible to identify the source of the error in the model, correct the model, and try again, as illustrated in Figure 1.1.
Several model-checking techniques can be used to solve real-world problems:

- **Kripke structure:** the basic idea is to generate a graph containing the reachable states of the system represented as nodes and the state transistions as edges. Once nodes are labelled with atomic propositions holding at each state, this graph is called *kripke structure*. For

small systems, checking if a Kripke structure is a model of the specification is very practical. However, for large systems (for instance, with many concurrent parts), global state transition graph turns out to be too large to handle. Further information about Kripke structures can be found in [12].

- **Symbolic model checking:** instead of explicitly building a Kripke structure as a graph, the main idea is representing the behavior of the system in a symbolic way. The feature of symbolic model checking consists in representing of sets of states of the system in implicit form, rather than having each global state explicitly represented (node of Kripke structure). Symbolic model checking can handle much larger models this way. There are several techniques that can be used with symbolic model checking, such as *Ordered Binary Decision Diagrams* (BDD's). A BDD is a canonical form for boolean expression, to represent the characteristic functions. Symbolic representation using BDD's has greatly increased the size of the verifiable systems, although many realistic systems are still too large to be handled. For further information about BDD's see [12].

- **Bounded model checking:** is a Propositional Satisfiability (SAT)-based technique where a system is unfolded *n* times and encoded as a SAT problem to be efficiently solved by a CNF(Conjunctive Normal Form)-based SAT solver. A satisfying assignment returned by the SAT solver corresponds to a counter-example of some length. This technique improves the scalability of symbolic model checking and it is currently used in many model checkers. Additional information about bounded model checking is available in [6].

- **Partial order reduction**: this technique can be used (on explicitly represented graphs) to reduce the number of independent interleaving (all the events in a single execution that are arranged in a linear order are known as *interleaving events*) of concurrent processes that need to be considered. As a result, the number of states needed for model checking is reduced. When a specification can not distinguish between different interleaving sequences, only one needs to be analyzed. Further information can be found in [8].

- **Abstraction:** the main idea is reducing the complexity of the system by modeling a simplified systems. Abstraction eliminates irrelevant, leading to simpler finite models expressive enough. Usually, the simplified system does not satisfy exactly the same properties as the original system. Hence, a process of refinement may be necessary. Generally, it requires the abstraction to be checked. However, most often, the abstraction is not complete (not all true properties of the original system are true in the abstraction). Thus, the simplified system loses precision regarding the original system. Additional information about abstraction can be found in [11].

- **Counter-example guided abstraction refinement:** due the disadvantage of abstraction, this technique starts by checking with a imprecise abstraction and iteratively refines it. When a counter-example is found, the tool analyzes it for feasibility (i.e., if the violation is genuine or the result of an incomplete abstraction). If the violation is feasible, it is reported to the user,

otherwise the proof of unfeasibility is used to refine the abstraction and model-checking starts again. Information about this technique is available from [10].

1.2.2 Theorem proving

Automated Theorem Proving (AST) (also known as *automated deduction*) deals with the implementation of computer programs showing that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). Given an appropriate formulation of a certain problem as axioms, hypotheses and a conjecture, an ATP system should be capable to solve the problem, however difficult it may happen to be. Due to this extreme capability, its application and operation sometimes requires the guidance of an expert in the domain of application to reduce the solving time. Hence, ATP systems are often used by domain experts in an interactive way. The interaction may be at a very detailed level (where the user guides the inferences made by the system) or at a very high level where the user specifies the intermediate lemmas to be proved on the path of the proof of a conjecture. There are several types of ATP systems, designed for several types of problems, operating in several ways and producing a range of different outputs. The evaluation of all types of ATP systems differs depending on the nature of the systems. Fundamental parameters by which ATP systems can be classified and how they are relevant to the evaluation schemes are discussed in [39].

1.2.3 Extended Static Checking

Software reliability adds to the risk of the overall system reliability. As systems have grown larger and more complex, functionality in mission-critical and safety-critical systems is more often exclusively controlled through software and consequently, the size and complexity of software has also been increasing. The more complex and larger software is, the harder it is to check for reliability and correctness. *Static analysis* (or *static checking*) is a technique capable of improving the quality and reliability of embedded systems software. Integrating static-checking tools and techniques into the development process can yield significant reductions in development testing and field failures. However, integrating static checking into a development process can be difficult, especially in large projects, with a reasonable amount of lines of code.

Static checking is a methodology of detecting errors in a program source code (or object code) without actually executing it. The basic idea is marking the areas where potential errors occur or may occur. The main advantage of using static code analyzers lies in the possibility of considerable cost saving in fault detection and elimination. The earlier an error is determined, the lower is the cost of its correction. There are several ways of applying static checking. Compilers are a common example where static checking is used. Checking for type errors or non-initialized variables are some features performed by a compiler that static checking provides. This methodology does not take many efforts considering the cost of running the tool. However, the degree of error coverage obtained is reduced.

Extended Static Checking (ESC) [19] is a broad name for a set of techniques for statically

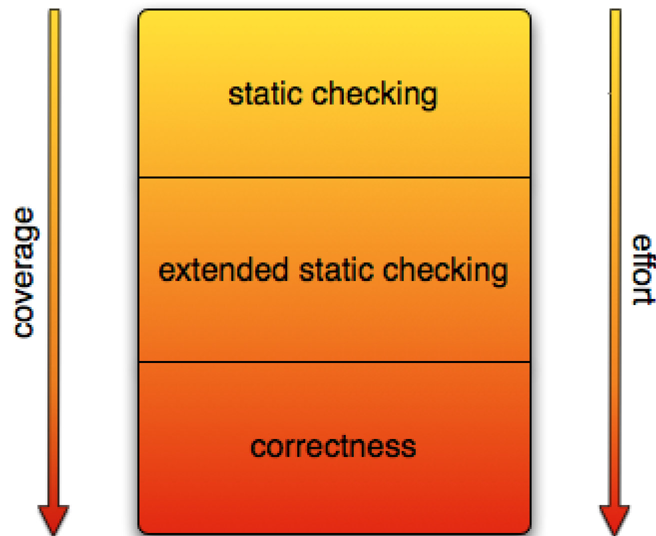


Figure 1.2: Static checkers comparison regarding error coverage and effort

checking the correctness of several program constraints. It can be thought of as an extended form of type checking: it provides higher ability of errors detection. The aim of ESC is evaluating the semantics of programs, providing several features such as:

- static warnings about errors usually detected at runtime by programming languages (null pointers, boundary errors, division by 0, etc);
- static warnings about synchronization errors in concurrent programs (dead locks, race conditions, etc);
- violation of design conditions: using an annotation language, the programmer specifies design decisions (pre-conditions, post-conditions, invariants) that will be checked by an extended static checker.

In fact, finding common errors mentioned above is the main goal of ESC. Hence, this verication technique does not provide total errors detection. Unlink ESC, the aim of *correctness* is proving the functional correctness of a given program. The higher degree of error detenction, the higher effort is required. Figure 1.2 illustrates a comparison between the three verification techniques on two important characteristics: the error coverage and effort required by each technique.

Several languages employ extended static checking such as JAVA (using Java Modeling Language (JML) for annotation and ESC/JAVA as an extended static checker), Haskell (ESC/Haskell) or Spec#. However, ESC can also be used with Point-free (PF) notation and relational algebra [34].

1.3 Problem Description

1.3.1 The UBIFS File System

I/O operations can often take a long time. To speed up this kind of operations a buffer allocated in the main memory is used. Buffers of this kind used by file systems are known as disk caches. They work by storing the most recently accessed data from the hard disk. Hence, when new data needs to be accessed, the file system first checks if the data is placed in the cache before reading it from the hard disk. The reason for this procedure lies in the access speed, i.e, accessing data from the RAM is faster than accessing from the hard disk. Therefore, disk caching can significantly increase performance.

Although disk caching improves performance, there is some risk involved. If a system crash occurs before the buffers have been written to disk, the last changes made to the data will be lost, and that could cause the system to behave in an inconsistent way after reboot. For instance, deleting a file on a Unix file system involves:

1. Removing its directory entry;

2. Marking space for the file and its node as free in the free space map.

If a system crash occurs between both steps, there will be an orphaned inode and therefore the allocated memory for the ”deleted” file will remain unavailable (memory cannot be released). On the other hand, if only the step 2 is performed first before the system crash, the file will be marked as free (before it is actually deleted) and it will possibly be overwritten. Detecting and recovering from an inconsistent state requires a complete scan of all data structures and this operation can take a long time if the file system is large. A solution is to maintain a log of the changes intended to make, ahead of time. This technique is called *journaling*. In a journaled file system, recovery simply entails reading the journal (log) from the file system and replaying the changes until the file system reaches a consistent state.

The Unsorted Block Image File System (UBIFS) [25] is a journaled file system developed by Nokia engineers with help of the University of Szeged, that works on top of UBI [20] devices (which is a wear-leveling and volume management system for flash devices). The data organization of UBIFS complies with the Linux virtual file system (VFS) specification: a node is a data structure that combines stored metadata and file data, and each node has a key associated (the Flash Store structure). Finding matching nodes for a given key by sequentially scanning the entire flash memory takes too long, hence UBIFS holds a data structure for mapping keys to addresses in the flash memory (called the RAM Index) that quickly finds nodes given a key, and also a similar data structure on flash for rebuilding the indexes (called the Flash Index). To limit the changes to the flash index, after updating the RAM Index, UBIFS saves the recent changes in a Journal (log) instead of updating the Flash Index immediately. The journal contains the addresses to the new data written to the flash store that have not yet been added to the flash index. Figure 1.3 illustrates the main data structures of UBIFS.

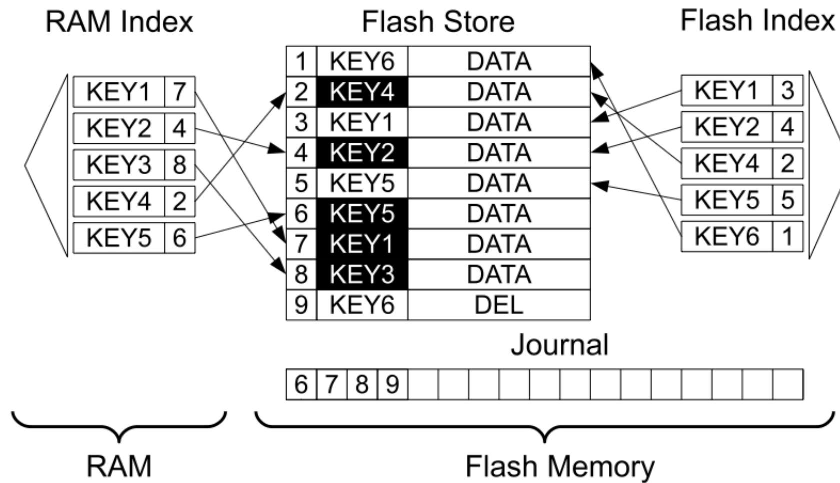


Figure 1.3: RAM index, flash index, flash store and journal
[36]

In this way, when a crash occurs, RAM index can be easily restored based on information from flash index, flash store and journal. When the addresses are replayed (restored to RM index), they usually are stored as new addresses (for the same keys) such as address 6 in Figure 1.3. The exception goes to address 9 that contains a deletion entry which means that KEY6 must be deleted from the index.

1.3.2 The State of the Art and Related Work

Joshi and Holzmann's mini-challenge had a major impact beyond the scientific community. In response to the mini-challenge, several file systems were (and currently are) cases of study. *Andy Galoway et al* report in [28] a case of study in modeling and verifying the VFS. The aim of the developed work is to evaluate the viability of retrospective verification of a VFS implementation using model-checking technique. In the paper, the authors present how to extract an executable model of the Linux VFS implementation, validate the consequent model by employing the simulation capabilities of SPIN, and analyse it for adherence to data integrity constraints and deadlock freedom using the SMART model checker.

Andrew Butterfield et al report in [9] the construction of formal models of NAND flash memory based on a standard for this kind of devices. The model is intended as a key part of the mini-challenge, involving the development of a verified file store system based on flash memory. The construction of the model involves building a highly reliable flash file store to use in space-flight missions, capturing the internal architecture of NAND flash devices. The article focuses on mechanising the state model and its initialization operation, which presents most of conceptual complexity.

The authors of [18] also contributed to the grand challenge by formally verifying an abstract model of the Intel® Flash File System Core. In the context of the mini-challenge, the paper focuses on the integration of different formal methods and tools for modeling and verifying of an abstract file system. High-level specifications are combined with mechanical verification tools into a tool-chain

that involves Alloy Analyzer for model-checking, Vienna Development Method (VDM) for specification and testing and High Order Logic (HOL) for theorem proving.

Andreas Schierl et al [36] developed a formal abstract specification for the UBIFS flash file system. Formal specifications were developed for the core components of the file system: the inode-based file store, the flash index, its cached copy in the RAM (RAM index) and the journal to save the modified files. Based on these data structures, the authors of the paper also wrote an abstract specification of the interface operations of UBIFS and proved some of the most important properties correct using KIV.

1.4 Aims

Due to its dimension and complexity, the UBIFS was found to be a good candidate to this comparative work. The first aspect to analyze is how KIV can be translated to Alloy, i.e., how data structures and operations are represented in Alloy. The main idea is to translate the model as literally as possible into Alloy bearing in mind the following questions:

- since in KIV the model is correct, will counter-examples be found?
- what vulnerabilities could the model have in Alloy that were not found in KIV?
- does the model take advantages of the tool?

Finally, it would also be interesting to compare both tools in several aspects such as language, features and usability (user interface). The main idea is to provide the user experience that might contribute to future and related work.

1.5 Document Structure

The next Chapter introduces the formal tools involved in this work that is particularly usefull for readers unfamiliar with KIV and Alloy. Chapter 3 presents the development process. It begins by describing in natural language the data layout of the file system and the underlying operations. The description of the file system is followed by the specification in KIV, illustrating some data types and operations. This Chapter ends with the corresponding specification in Alloy, pointing all the important aspects of mapping between KIV and Alloy. Chapter 4 presents some concluding remarks of this project and gives directions for future work. The concluding remarks are divided in several topics to clarify the conclusions of the comparative work. Future work addresses not only the goals not achieved but also what could be done in the area.

Chapter 2

Tool Background

2.1 KIV

The Karlsruhe Interactive Verifier (KIV) [1] (KIV) system is a tactical theorem prover [15] developed by *M. Heisel et al* at University of Karlsruhe, Germany, that can be applied in several areas. Some of these areas are:

- the development of safety critical systems from formal specifications;
- verification of safety requirements and correctness of implementations;
- semantic foundations of programing languages from a specification of the semantics downto a verified compiler;

KIV is a powerful tool capable of providing strong proof support (automation, heuristics, simplification, etc). KIV can hold large scale formal models by efficient proof techniques, multi-user support and user-friendly graphical user interfaces. It has been used in several projects available for download at the KIV Internet site [2]. For further details about a few KIV projects see [21] and [23].

The KIV system provides a functional programming language called Proof Programing Language (PPL) to implement both the formal and the informal aspects of program development. Proof tactics are represented as PPL programs constructing proofs in the underlying logical formalism. The main advantage of this representation of proofs (and specifications) is that they can be stored and checked by the user and the system itself, enabling the re-use of proofs, specifications and verified components.

KIV is now in version 5.1 and it is only available for *Linux* platform. For other operating-systems a virtual machine is available.

[1] KIV home page: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv
[2] http://www.informatik.uni-augsburg.de/swt/projects/

2.1.1 Specification Language

KIV adopts a combination of *higher-order* algebraic specifications and *dynamic logic*. The structure of a system can be described using algebraic specifications. These are built on top of elementary specifications. Basic specifications are divided in three parts: a description of the signature, axioms and principles of induction. A basic specification is the textual content between the keywords `specification` and `end specification`.

A signature of a specification can be described by the declaration of:

- **sorts –** enumerates all sorts of the specification;
- **constants –** indicates all constants of the specification; they are used in axioms;
- **functions –** functions are operations with a certain number of input arguments that return a result;
- **predicates –** predicates are operations that specify the validity of a set of properties;
- **variables –** determines all variables of the specification; they are used in axioms;

This is a partial example of a specification to represent an heap [3]:

heap =
enrich Ref, cell, nat **with**
sorts heap;
constants $\emptyset$: heap;
functions

. [.]	:	heap $\times$ Ref $\times$ cell	$\rightarrow$	heap	;
. [.]	:	heap $\times$ Ref	$\rightarrow$	cell	**prio** 2;
new	:	heap	$\rightarrow$	Ref	;

predicates . $\in$. : Ref $\times$ heap;
variables H, H0, H1, H2: heap;

axioms

heap **generated by** $\emptyset$,];

...

In-empty : $\neg$ r $\in$ $\emptyset$;

In-insert : r $\in$ H[r0, ce] $\leftrightarrow$ r = ro $\vee$ r $\in$ H;

At-same : H[r, ce][r] = ce;

...

[3] Example available in KIV 3.0 distribution package

end enrich

The syntax of the KIV language has a few similarities with the *higher-order* algebraic specifications. An example of this is the ability of using special characters in the specifications, such as greek letters or logical symbols. Thus, specifications became more legible to the user. The second part of a basic specification is the definition of principles of induction. They are specified after the keyword `induction` and it consists of a `generated by` clause. Principles of induction are used to generate elements of the data types and they are followed by axioms. These are placed after the keyword `axioms`. Not all basic specifications contain principles of induction and axioms, they may merely contain its signature.

Besides the basic specifications, KIV language provides several others, such as:

- **Generic specifications –** unlike the basic specifications, *generic specifications* are composed by a parameter. They are commonly used to define sets, lists or arrays of the parameter data type.

- **Data specifications –** *data type specifications*, typically, are used to define data types and their constructors, i.e., *enumeration types*, *tuples*, *variant records* or *natural numbers*. These are useful when a data types can acquire different definitions or when there is, for each argument of the constructor, a function that selects each field of the data type.

- **Union specifications –** are used to combine several specifications. *Union specifications* begin with the keywords `union specification` followed by the name of the specifications, separated by the character "+". Trivially, signatures, axioms and parameters of union specifications are the union of the signatures, axioms and parameters of the subspecifications.

- **Actualizations –** generic specifications are instantiated by *actualizations*. Actualizations star with the instantiation of a *parameter specification* indicating that the components (parameter, sorts, variables, operations, etc) of the generic specification must be updated. The second step consists of instantiating the *actual specification*. The specification is updated in the third step: mapping the parameter specification to the actual specification. This operation is called *morphism*. An example of an actualization extracted from [1], an example of such specification is given below.

  keylist =
  actualize list-dup **with** key **by morphism**
  list $\rightarrow$ keylist; elem $\rightarrow$ key; ...
  end actualize

  The generic specification `list-dup` is updated by the specification `key` where the selected parameters of `list-dup specification` (`list`, `elem`...) are replaced by `keylist` and `key` respectively.

- **enrichment –** *enrichments* are used to enlarge a small specification by adding new sorts and operations. The parameter of the enriched specification is the parameter of the small specification. It cannot be enriched by parameter operations. All the content added to the enriched specification is delimited by the keywords `enrich` and `end enrich`.

- **renaming –** a *renaming* specification is used to rename sorts and operations of a specification. It starts with the keyword `rename` and the keywords `by morphism` are followed by a list of renamings according to the description in the case of updates. The specification ends with the keywords `end rename`.

Dynamic logic is an extension of *higher-order Logic* used for correctness proofs. It allows the formalization of programs properties such as correctness specifications, equivalence programs, programs synthesis from specifications, etc. Correctness specifications in KIV are written as theorems. Further details about correctness proof can be found at Section 2.1.2.

2.1.2 Proof suport

The KIV system provides an interactive deduction component based on proof tactics in order to suport specification validation and design as well as program verification. An high degree of automation is combined with an interactive proof environment. The proof support is based on:

- sequent calculus with proof tatics: simplification, lemma application, induction for first-oder reasoning;

- intuitive proof strategy based on symbolic execution with induction. Induction is used for the verification of implementations with imperative programs using *dynamic logic*.

In order to automate proofs, the KIV system provides a set of heuristics: induction, simplification, etc. The purpose of the heuristics is the application of tactics and reduction of goals to subgoals.

Heuristics can be chosen and changed any time during the proof. For instance, if all heuristics fail, the user is allowed to:

- select another tactic or heuristic;

- introduce lemmas;

- do backtracking: if a proof becomes stalled it is possible to backtrack or restart at the point of the choosed proof and try another;

A complete proof for a given formula φ means to reduce φ in the formula $\top$. To achieve this kind of simplification, simplifier rules should be used. KIV simplifier handles thousands of rules very efficiently by compiling them to executable functional programs. The structure of a formula facilitates the understanding of its meaning, therefore the KIV simplifier always maintains this structure. Rewrite and simplification rules can be chosen directly by the user for several tasks.

Usually, in software verification, it is harder to analyzes a failed proof than to conduct a proof. Analyzing failed proof attempts that may indicate errors in specifications, programs or lemmas may raise some question. In this context, KIV provides several proof engineering techniques in order to support the iterative process of (failed) proof attempts, error detection, error correction and re-proof. Another feature of KIV system consists in using a strategy for proof reuse. When a subgoal can not be proved, counter examples are automatically generated. Through the counter example, the user can be directed to the earliest point of failure. Thus, the user can easily decide if the decisions made in the proof are incorrect or if there is a flaw in the specification and later re-proving the goal.

The correctness management in KIV assures consistency in changes or deletions of specifications, modules and theorems, and that the user can do proofs in any order. It ensures that: only the minimal number of proofs are invalidated after changes, there are no cycles in the proof hierarchy and finally all used lemmas and proof obligations are ascertained (in some subspecification).

2.1.3 User interface

This section describes the user interface provided by KIV. Following the description of the authors of KIV, the information found in this section was partially quoted from the KIV home page[4]. KIV provides a powerful graphical user interface including several features. In order to manage large applications, the user interface was designed to allow easy access to KIV for first time users. Figure 2.1 illustrates the appearance of the user interface of KIV.

The top-level object of a KIV project, the development graph, is displayed using daVinci, a graph visualization tool which automatically arranges large graphs conveniently. Figure 2.2 shows a graph visualization of a small example[5].

The theorem base, which is attached to each development node, is arranged in tables, and context sensitive popup menus are provided for manipulation. While proving a theorem, the user is able to restrict the set of applicable tactics by selecting a context, i.e. a formula or term in the goal, with the mouse. This is extremely helpful for applying rewrite rules, as the set of hundreds of rewrite rules is reduced to a small number of applicable rules for the selected context. Proofs are represented as trees, where the user can click on nodes to inspect single proof steps. Figures 2.3 and 2.4 illustrate respectively a proof menu and a proof tree of a small example[6].

In large applications, the plentitude of information may be confusing. Therefore, important information is summarized, and more details are displayed on request. Different colors are used to classify the given information. Additionally a special font allows the use of a large number of mathematical symbols. Figure 2.5 contains the information of a proof provided by KIV.

KIV automatically produces LaTeX documentation for a project on different levels of detail. Specifications, implementations, theorem bases, proof protocols, and various kinds of statistics are pretty printed. The user is encouraged to add comments to specifications, which are also used to automatically produce a data dictionary. As several users may work simultaneously on a large project, the

[4]http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/
[5]Example available in KIV 3.0 distribution package
[6]Example available in KIV 3.0 distribution package

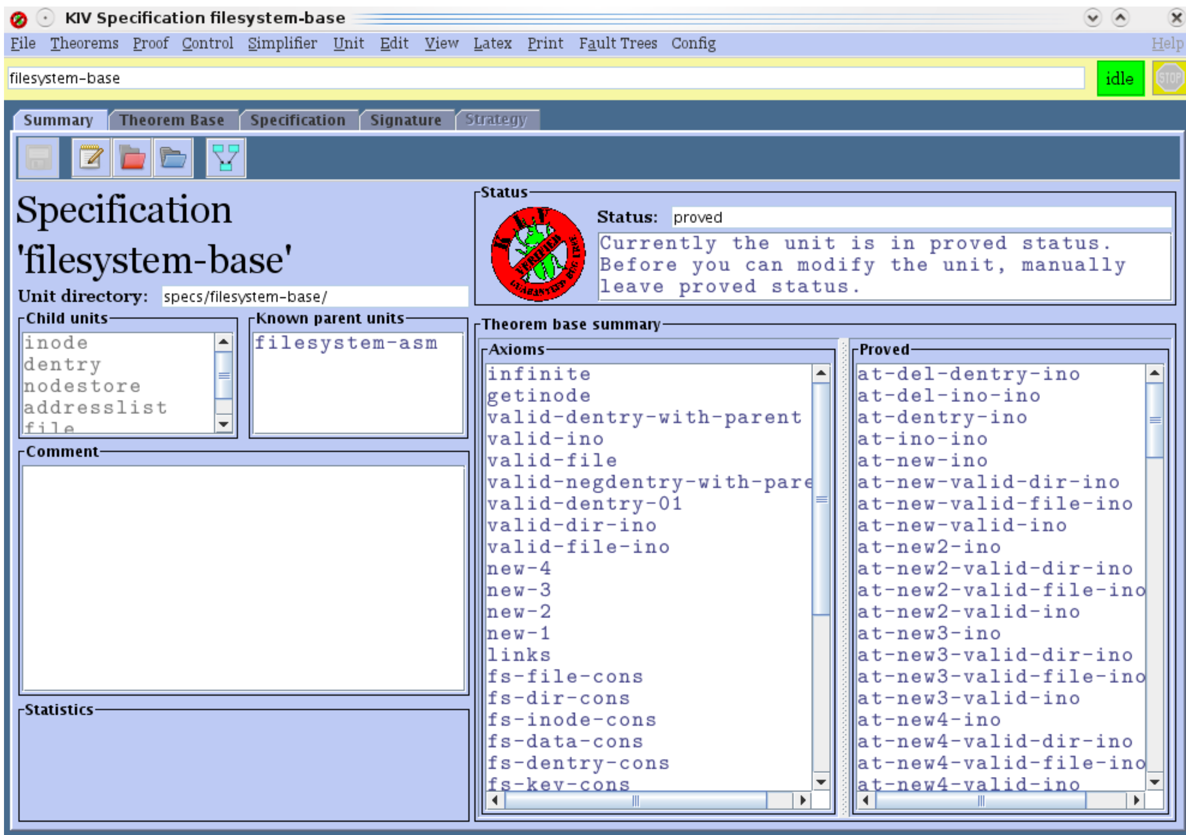


Figure 2.1: Graphical user interface of KIV

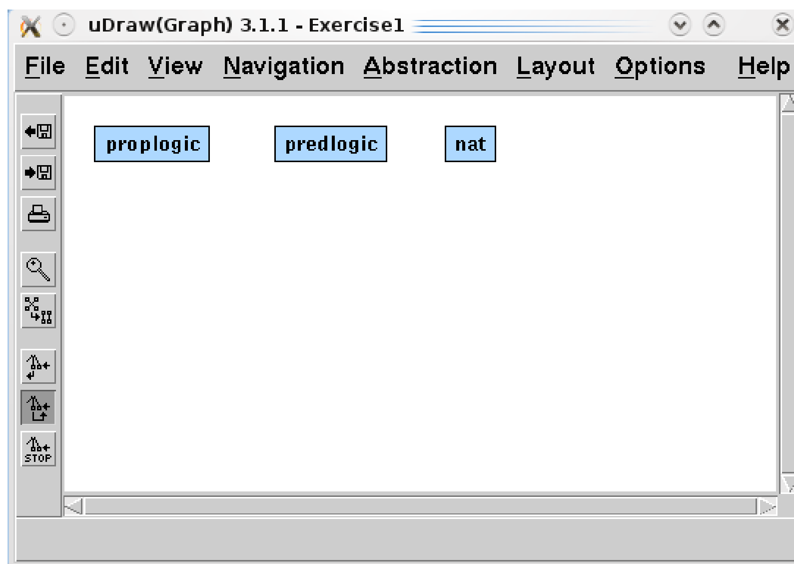


Figure 2.2: Graph visualization of KIV

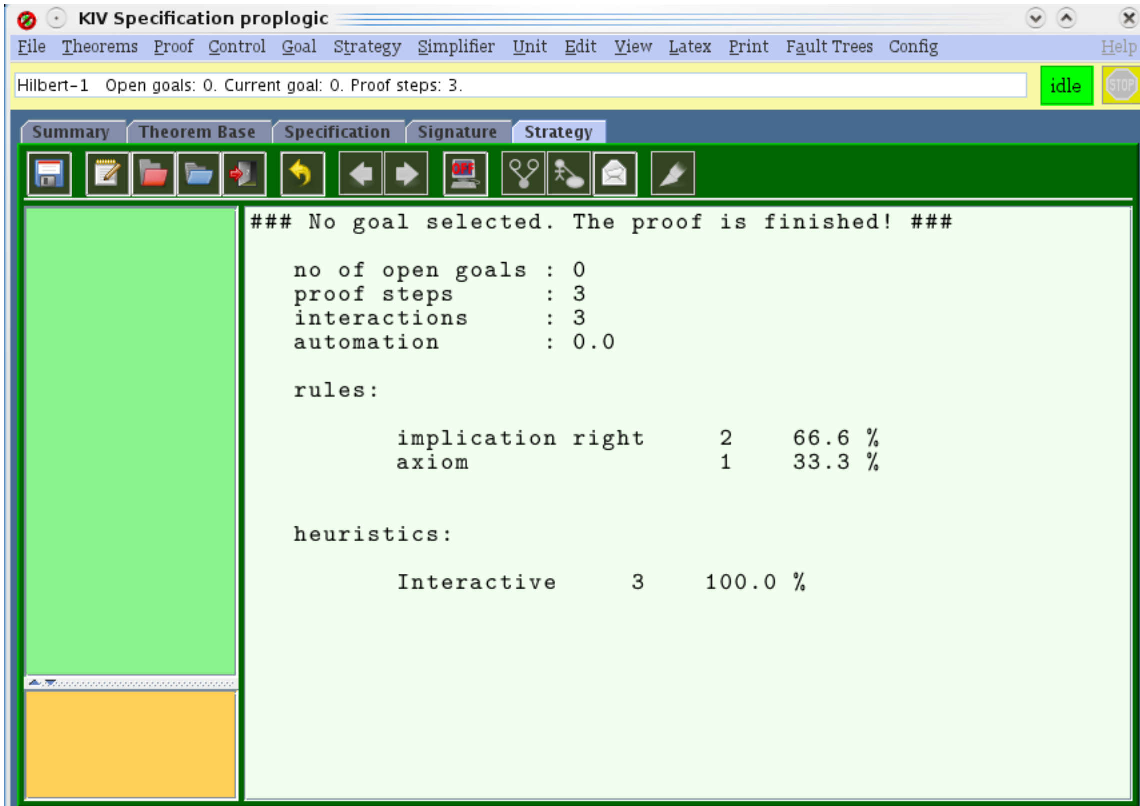


Figure 2.3: Proof visualization of KIV

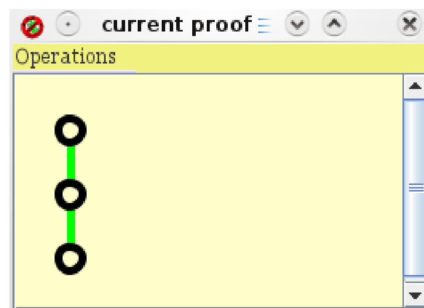


Figure 2.4: Proof tree of KIV

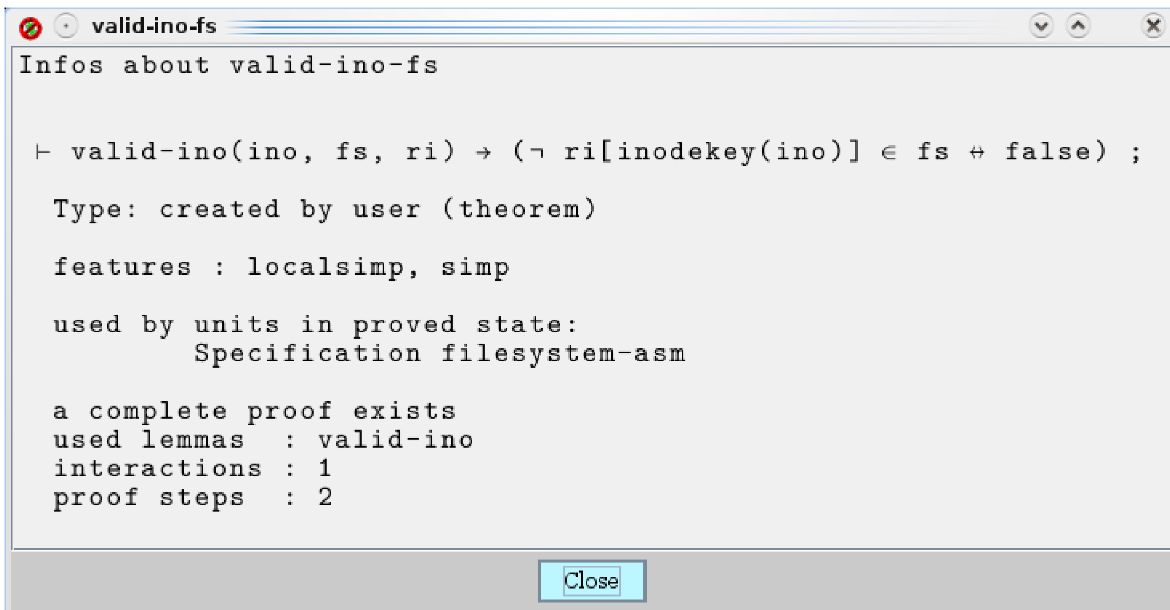


Figure 2.5: Information of a proof

documentation facilities of KIV are very important. The automatically extracted information can also be included into reports. All the KIV specifications used in this thesis were automatically generated by KIV.

2.2 Alloy

Alloy[7] is a lightweight modelling language developed at MIT under the guidance of Daniel Jackson and is used in software design: it allows the creation of a model that specify the constraints and the behaviour of a software system. Alloy also provides a modeling tool based on first-order logic, the *Alloy Analyzer*, used to check Alloy specifications for correctness.

2.2.1 Specification Language

Alloy is a declarative specification language, expressive enough to describe the structure of complex systems, but simples enough to be processed by fully automated analysis. It allows the user to specify the behavior of a system without describing its internal steps and to avoid precipitated implementation decisions.

The specification language consists of a small number of basic constructs: signatures, relations and operators, predicates, functions, assertions and facts.

[7]Alloy home page: http://alloy.mit.edu/community

Signatures

In Alloy ”Everything is a relation” **??**, i.e., the Alloy universe consists of atoms and relations. An atom is called signature and signatures represent data types, e. g.

```
sig Name{}
```

`Name` is a signature that represents ”something”. A signature can be abstract and/or can be a super-type of others signatures, similar to the concept of inheritance in the Object Oriented Paradigm. For instance:

```
abstract sig Key {
   ino: Int
}

sig Dentrykey extends Key {
   name: Name
}
```

`ino : Int` is a field of the signature *Key* and is a way to declare a relation, where ino is the name of the relation. sig *Dentrykey extends Key* declares a dentrykey as a sub-type of a key. Note that *Dentrykey* also has the field ino from *Key*. A relation can also be declared as a map, for instance,

```
sig UBIFS {
    fs : Address →lone Node,
     ...
```

where `fs` is, besides a field of *UBIFS*, a relation of arity 3 that its mapped by *UBIFS* and it also maps Adresses to one or zero Nodes. Quantification in Alloy is denoted by multiplicity factors descreibed in the list given below.

- lone: zero or one
- set: zero or more
- some: one or more, represents the quantifier $\exists$
- one: exactly one
- all: represents the quantifier $\forall$
- no: zero

By default one is assumed, e. g. `name: Name`.

Constraints

Every system has its constraints: non-null values, positive values (for instance, an inode can not have a negative size), etc. In Alloy, there are several ways to define constraints. *Facts* are constraints that are always holded, for instance, the size of an inode that must be a non-negative number. In Alloy, it can be written as follows:

```
fact Notnegative {
  all n : Node | n·size ≥ 0
}
```

Besides facts, constraints can also be declared in signatures. For instance, dentrykeys only identify dentry nodes. In Alloy it is defined as follows:

```
sig Dentrynode extends Node {
   name : Name,
   dino : Int
}{ key in Dentrykey}
```

There are several constraints that only make sense in a certain context. In this case, they are defined using *predicates*, i.e., they may be represented by predicates.

Predicates

Besides constraints, *predicates* may also represent operations over a signature. A predicate may add constraints or modify the state of a signature, without producing a result. An example of a predicate is given below:

```
pred valid_ino [ino_ : Int, fs : Address →lone Node,
                ri : Key →lone Address]
{
    one ink : Inodekey | ino in ink·ino ⇒
        ink in ri·dom and ri[ink] in fs·dom
    ino_ ≠ 0
}
```

In the case above, the predicate `valid_ino` specifies constraints over `ri`. It ensures the validity of an inode, which is identified by its key, in the RAM Index: all keys in RAM Index should be in file store and the identifier must be different from 0.

Functions

Similarly to predicates, functions can also represent an operations over a signature. However, unlike predicates, functions produce a result and they do not modify the state of signatures. An example of a function is given below.

```
fun dentrykeys [ri : Key →lone Address, ino_ : Int] : set Dentrykey
{
    key : ri·dom | key·ino in ino_
}
```

The function dentrykeys returns a set of Dentrykeys from the domain of ri that the field ino has the same value of ino_.

Assertions

Assertions are constraints which the model must follow and the command check is used by the analyzer to search for counter-examples within scope:

```
assert datakeys_dir {
  all u : UBIFS, ino : Int | valid_dir_ino[ino, u·fs, u·ri] and datanode_cons[u·fs, u·ri] ⇒ no datakeys[u·ri,ino]
}

check datakeys_dir for 4
```

The assertion means that for all file systems and integer numbers, if the predicates valid_dir_ino [ino, u.fs, u.ri], datanode_cons[u.fs, u.ri] are satisfied, there are no keys in the RAM index of the file system, that are identified by ino. The command check will be described in Section 2.2.2.

2.2.2 Alloy Analyzer

Alloy Analyzer is a modeling and verification environment. On the one hand, it provides an Alloy editor to write all needed specifications, on the other hand Alloy Analyzer is equipped witha SAT-solver, capable of showing visually the results of the model's analysis by generating a meta-model, finding counter-examples or instances of certain predicate. Figure 2.6 illustrates the appearance of the Alloy user interface.

Considering the screenshot of the Alloy Analyzer in 2.6, a metamodel was generated based on the defined signatures. In the current example, Alloy Analyzer generated a metamodel of a directory entry that is illustrated by Figure 2.7.

In order to find counter-examples and/or instances of a predicate, Alloy enables the use of commands like run and check. As the name suggests, run command is used to run a predicate as follows:

```
run new_inodekey for 2
```

The integer value defines the scope of the search for instances. In the current example, *new_inodekey* is a function that returns a new inodekey. If the the predicate (or function) is valid,

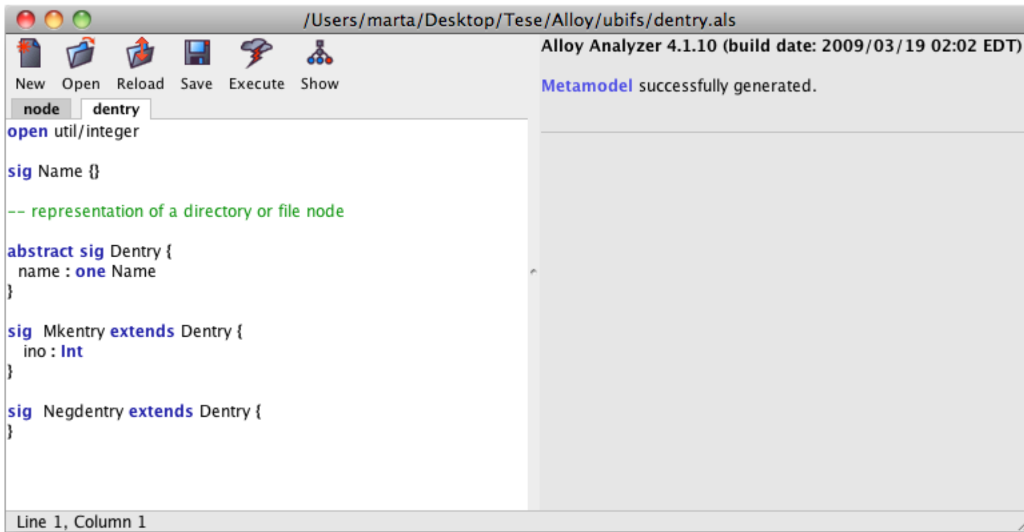


Figure 2.6: User interface of Alloy Analyzer

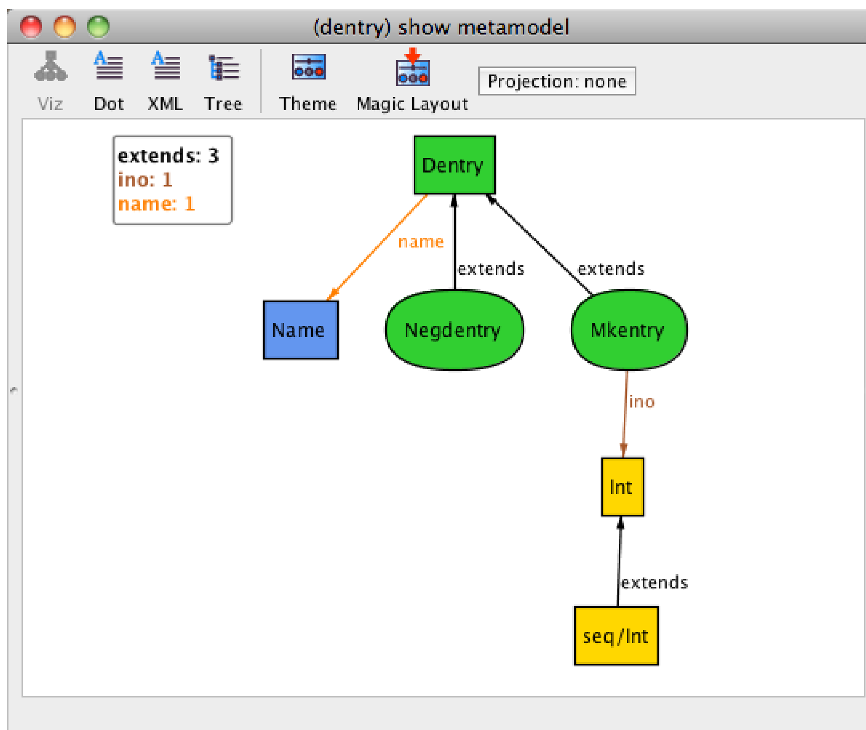


Figure 2.7: Metamodel of a directory entry generated by Alloy Analyzer

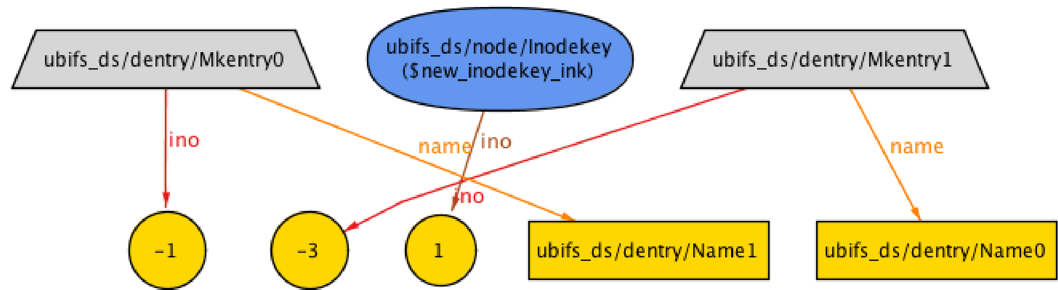


Figure 2.8: Instance found of the function new_inodekey

Alloy Analyzer shows the instances found shown in Figure 2.8. If no instance can not be found, Alloy Analyzer classifies the predicate (or function) as invalid.

The command `check` is used to check if the model follows a given assertions. It is used by the analyzer to search for counter-examples within scope, as shown in the example given in Section 2.2.1.

If no counter-example is found, *Alloy Analyzer* produces an output similar to the following:

```
Executing "Check ri_fs for 3 but 2 UBIFS"
   Solver=sat4j Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20
   8990 vars. 559 primary vars. 23581 clauses. 293ms.
   No counterexample found. Assertion may be valid. 22ms.
```

The lack of counter-examples does not assure total correctness of the model. *Alloy Analyzer* can falsify an assertion by finding counter-examples within a certain scope. However, it cannot prove that a assertion holds in an infinit states space.

Chapter 3

Mapping between *KIV* and *Alloy*

In this chapter, shall be presented not only the abstract specification of UBIFS developed in [1] but also the transcription of the specification to the KIV specification and the re-writing in Alloy and the differences between the both tools. There are many differences between both tools. The main difference between both tools is that KIV is a theorem prover and Alloy is a model-checker. While a theorem prover is used to produce formal proofs for theorems, a model-checker searches for counter-examples to a given constraint of the model based on the description of both languages, the mapping will be divided in several sections: data types, axioms/invariants, operations and correctness.

Data types in KIV are defined using several kinds of specifications and constructors. In Alloy, data specifications are replaced by signatures containing, in their body, relations (or fields). Relations represent the fields of the constructor used to define a certain data type. Sections 3.2.1 and 3.3.1 contain further information about the mapping between specifications and signatures.

The syntax used to define operations in KIV is a variant of Abstract State Machine (ASM) syntax. Therefore, operations are specified using ASM rules. Section 3.2.3 contains more detailed information about operations in KIV. In Alloy, ASM rules are translated in predicates. Hence, each operations of the file system is represented by a predicate. Further information about operations in Alloy is given in the Section 3.3.4.

The main invariants of the UBIFS are represented as predicates. However, they may resort to the use of functions. In each language, the invariants are defined using two completely different techniques. KIV provides axioms that are used to define the behavior of functions and predicates. The difference between both is the associated keyword: `functions` and `predicates`. These display the signature of the functions (respectively predicates) used in a certain specification. According to Section 2.2.1 and Section 2.2.1, in Alloy predicates and functions are directly translated using the corresponding concepts provided by the tool. Additional information is found in the Sections 3.2.2 and 3.3.3.

In KIV, theorems are statements used to prove termination of operations and correctness assertions. Theorems are also used to that some axioms are, in fact, invariants of the operations. In Alloy, theorems are replaced by assertions. These are used to check the existence of counter-examples for the specified invariants. Sections 3.2.4 and 3.3.5 explain the process of correctness and its

transition to model-checking.

In order to improve the readability of the developed model, Section 3.1 will present a brief description of the UBIFS.

3.1 Abstract Specification

The data layout used in UBIFS follows the file system representation of Linux VFS. The main data structure used in UBIFS is *index nodes* (inodes). They denote objects such as files or directories and they are identified by an *inode number*.

Inodes store information about the represented object such as size, link count, modification date, permissions, etc. However the name of the object is not included. The association between names and objects is done by *directory entries* (dentries). They specify which object belongs to which directory under a concrete name. If an object is a file, its content is stored using *file inodes*. Each data inode (file inode) contains fixed-size data blocks named *pages*. When opening a file or directory to access its content, a data structure is used to manage the inode number and the current position within the inode. Therefore, UBIFS holds three variations of nodes:

- **inode nodes:** to store the information of an inode;
- **dentry nodes:** to represent a directory entry;
- **data nodes:** that holds a page with the content of a file

For each kind of node, there is a kind of key associated:

- **inode key:** that identifies an inode by its number;
- **dentry key:** that provides the inode number of the containing directory and the name of the file;
- **data key:** to specify a page of data of a certain file

Nodes and keys are mapped using two variants of a data structure: the main data structure (called *flash store*) that maps addresses to nodes and two similar data structures (*flash index* and *RAM index*) that maps keys to addresses. The journal of the file system is represented by a list of addresses. Therefore, the entire file system is composed by these four data structures: the flash store (*fs*), the RAM index (*ri*), the flash index (*fi*) and the journal (*log*). A directory entry is represented by the data structure *Dentry* and it comes in two variants. A *Mkentry* associates a file (refered by its inode number) to its directory parent (refered by its name). A *Negdentry* only contains the name of the corresponding directory.

Operations are used to apply changes to the contents os the file system. They can be grouped into inode operations, file operations and space address operations.

As the name suggests, inode operations work over inodes. They allow creating, renaming or deleting inodes of the file system.

File operations work with inode contents. These can be files or directory entries. File operations allow, for instance, to open files or directory entries.

Address space operations include all operations that access file contents. They are included in [36] to allow using abstract pages when handling file contents.

The following lists present the main implemented operations of the file system. They are represented using KIV signature for a better explanation of their behavior. The description of all the operations was partially quoted from [36].

The following list refers to inode operations:

- **create (P_INO; DENT, FS, RI, LOG) –** creates a new file whose name is specified by DENT, in the directory identified by P_INO. FS, RI and LOG are updated with the new entries created: an inode node for the created file, the corresponding directory entry and an inode node to increase the parent directory size by 1 to reflect the increased number of objects contained in the directory. DENT is returned as the the newly created dentry.
- **unlink (P_INO, DENT) –** is the "delete" operation. It removes the file referred to by DENT from the directory P_INO. If the dentry was the last link to the referred file, the inode and file contents are also deleted, otherwise only the dentry is removed. DENT is returned as a negative dentry.
- **link(OLD_DENT, NEW_INO, NEW_DENT) –** creates a hard link to the file referred to by OLD_DENT, placed in the directory NEW_INO and named as given by the negative dentry NEW_DENT. Returns the newly created dentry in NEW_DENT.
- **mkdir(P_INO, DENT) –** creates a new directory in P_INO, with the name given in the negative dentry DENT. The newly created dentry is returned in DENT.
- **rmdir(P_INO, DENT) –** removes the (empty) directory referred to by the dentry DENT located in the parent directory P_INO. DENT is changed into a negative dentry.
- **rename(OLD_INO, OLD_DENT, NEW_INO, NEW_DENT) –** moves the object (file or directory) referred to by OLD_DENT from directory OLD_INO to directory NEW_INO, changing its name to NEW_DENT.name. If the object referred to by NEW_DENT exists, it has to be of the same type (file or directory) as OLD_DENT, and it is overwritten (i.e. deleted).
- **lookup(P_INO, DENT) –** checks for the existence of a object named DENT.name in the directory P_INO. If it exists, the dentry is returned in DENT, otherwise a negative dentry is returned.

For inode contents, the following file and address space operations are:

- **open(INO, FILE) –** opens the file or directory given in INO, and returns a new file handle in FILE.
- **readpage(FILE, PAGENO, PAGE) –** reads the page with number PAGENO from the file referred to in FILE, and returns it in PAGE.

- **writepage(FILE, PAGENO, PAGE) –** writes the data from `PAGE` as new page numbered `PAGENO` to file `FILE`.
- **truncate(FILE, PAGENO) –** sets the file size of the file referred to in `FILE` to `PAGENO`, deleting all pages beyond.
- **readdir(FILE, DENT) –** Returns the next object of the directory referred to in `FILE`, or a negative dentry if no further file or directory exists. The (positive or negative) dentry is returned in `DENT`, and the position stored in `FILE` is increased to return the next object at the next call.

For the replay process and garbage collection, the operations are:

- **gc(FS, RI, FI, LOG) –** represents the *garbage collection* operation. It stores additional metadata by duplicating the content of `FS, RI, FI` and `LOG`, ensuring the consistency of the metadata.
- **commit(RI; FI, LOG) –** stores, in the flash index, the changes made since the last commit. The content of `RI` is copied to `FI` and `LOG` is cleaned.
- **replay(FS, FI, LOG; RI) –** implements the replay process. The RAM Index is restored by copying the content of `FI` to `RI` and applying all changes recorded in `LOG` to `RI`.

3.2 KIV Specification

This section presents an extract of the KIV specification of the UBIFS from [1]. Section 3.2.1 specifies the main data structures of the file system. Section 3.2.2 specifies the axioms defined that are used in correctness and Section 3.2.3 illustrates an operation of the file system.

3.2.1 Data Types

In KIV, most of data types are represented as data specifications. In order to specify the several variations of nodes (respectively keys) a data specification *node* (respectively *key*) is defined containing all the necesssary contructors, as depicted in Appendix A.1.1 (respectively Appendix A.1.2).

In the example of Appendix A.1.1 , `. .ino`, `. .directory`, `. .nlink` and `. .size` are infix function selectors available on the constructor `inodenode` in order to access its arguments. The keyword `with` followed by `inode?` is used to test if `inodenode` was generated by the predicate `inode`. The variables defined in the data specification are used in axioms, lemmas and theorems specifications.

Generic specifications are also used in the definition of the data types. It is the case of the generic definition of stores. *store* is the generic specification to represent the main data structures of UBIFS. Therefore, once *store* is a partial function that maps elements to data, it is used to represent the flash store. The corresponding specification is found in Appendix A.1.1.

In order to represent the four main components of the model, two variations of the *store specification* are used: one to define the mapping between addresses and nodes and another to define the mapping between keys and addresses. The corresponding specifications in KIV can be found in Appendix A.1.5 and Appendix A.1.6 respectively.

The journal is defined as a list of addresses. it refers to an actualization of the specification *list-dup*. This is an enriched specification of a specification of lists. The KIV specifications for log and *list-dup* can be found, respectivelly, in A.1.9 and A.1.7.

The toplevel specification is called *file system* and it only defines an invariant. Further details are approached in Section 3.2.2.
A graphical overview of the full project can be found in A.1.16

3.2.2 Axioms

Specifications are not only used to define data types but also functions, predicates and axioms. This is the case of the specification *filesystem-base*A.1.9. It contains basic operations (functions) and predicates for the file system. These predicates are understood as invariants (once they can specify consistency of the file system) or pre-conditions (used in the main operations or other predicates).

As said in Section 2.1.1, functions and predicates only specify the signatures (input and output arguments), i.e, they do not specify their behavior. It is defined by statements called axioms.

Basically, functions are used as auxiliary operations. Their output is used in the body of predicates and procedures (axioms). An example of a function is *getinode*. Given an inode number, a nodestore and a nodeindex, *getinode* outputs the existing inode in both data structures, identified by the inode number given as input. The inode returned is a variation of the constructor *inodenode*. The constructor used instead of *inodenode* was *mkinode*. The only difference between both is that *mkinode* associates to the node an inode number and *inodenode* associates a key. *mkinode* is defined in the data specification *inode*. The signature and body of the function *getinode* can be found in Appendix A.1.9.

Predicates are used to specify invariants and pre-conditions. Its definition is similar to the definition of functions. *fs-cons* is an example of a predicate that ensures the consistency of the flash store and RAM index:

- the flash store contains the RAM index (all keys and addresses of the RAM index belongs to the flash store);
- the keys of the RAM index match with the corresponding inodes of the flash store;
- the consistency of the three variants of they keys is ensured (by the predicates *fs-dentry-cons*, *fs-inode-cons* and *fs-data-cons*).

The two first conditions are ensured by the predicate *fs-key-cons*. The three auxiliary predicates (*fs-dentry-cons*, *fs-inode-cons* and *fs-data-cons*) ensure the validity of the corresponding inodes. The definition in KIV of these predicates can be found in the *filesystem-base* specification.

valid-dentry and *valid-file-ino* are two predicates that ensure, respectively the validity of a directory entry and the validity of a file. They work as pre-conditions of the operation *unlink* whose description can be found in Section 3.2.3.

The main invariants of the abstract specification that ensure the consistency of the file system (directly used in the verification) are:

- **fs-cons –** as mentioned before, ensures the consistency of the keys.
- **datanode-cons –** assures the consistency of the datakeys and datanodes, all the datanodes must represent valid files and the part number of its datakey must be smaller than its size.
- **nodekey-cons –** guarantees the association of one key to one node, all keys of the RAM index map the addresses of the corresponding nodes. *nodekey-cons* can be found in the *filesystem-base* specification.
- **store-cons –** assures the consistency of the addresses of the file system: all addresses stored in the journal and RAM index must be stored in the flash store.

The predicate *log-cons* is also an invariant of all operations, however its definition is not in the *filesystem-base* specification but it can be found in the toplevel specification A.1.12. The *replay* operation is needed to define *log-cons*, therefore its axiom can be found in the top level specification. Further information about *log-cons* can be found in the Section 3.2.4.

The remaining axioms of *filesystem-base* work as pre or post-conditions of the operations. They are also used in the body of functions and in the body of other invariants and/or predicates.

3.2.3 Operations

Specification *filesystem-asm* contains all the operations of the file system. The syntax is similar to the ASM notation, however, the operations are not executed atomically (as an ASM rule), the semantics of KIV's temporal logic only executes parallel assignments atomically. *FSOP* is the top level operation that selects suitable input data and calls one fo the operations of the file system. Operations are defined using procedures. The keyword *procedures* indicates the signature of all operations and the keyword *declaration* specifies its behavior (similar to predicates/functions and axioms). Appendix A.1.10 illustrates an extract of the top level operation and of the operation *unlink*.

Note that the condition

```
choose P_INO, DENT with (valid-dentry(P_INO, DENT, FS, RI) ∧ valid-file-ino
(DENT.ino, FS, RI)) in unlink ifnone skip
```

means that if `valid-dentry (P_INO, DENT, FS, RI)` and `valid-file-ino (DENT.ino, FS, RI)` are satisfiable, *unlink* is executed, otherwise the execution terminates. Hence, these two conditions are understood as pre-conditions of the operation.

In order to understand the procedure of the *unlink* operation, it is necessary to consider the definition of the involved data types. *Dentry*A.1.3 is the data type that represents the information of

a (existing or non-existing) file. Note that *dentry* and *datanode* are distinct data types. A *datanode* stores the content of an existing file in the file system while a *dentry* comes in two variants: a *mkentry* that identifies an existing file by its name and inode number and *negdentry* (negative dentry) that specifies a non-existing file by its name.

Retrospecting the definition of *node*A.1.1 and *getinode*A.1.9 and the definition of *unlink*, the procedure of the operation (which can be found in Appendix A.2.15) starts by declaring three variables:

- P_INODE – represents the inode of the parent directory (identified by P_INO)
- INODE – refers to the inode of the deleting file
- NODE – represents the updated inode node of the deleted file (the link count must be decreased by 1)

As said in the Section 1.3, removing a file from the file system means marking it as a deleted file. Therefore, the main data structures of UBIFS must be updated. The flash store is updated with three new entries:

- `[ADR1, dentrynode(dentrykey(P_INO, DENT .name), DENT .name, 0)]`: creates a *dentrynode* with destination inode number 0 to mark the file as removed
- `[ADR2, inodenode(inodekey(P_INODE .ino), P_INODE .directory, P_INODE .nlink, P_INODE .size - 1)]`: decreases by 1 the size of the parent directory (size is taken by the number of the contained objects)
- `[ADR3, node]`: adds `NODE` to the flash store

The RAM index is also changed. The key of the deleted file is removed and a new entry is added with the key of the updated inode node of the parent directory. If the number of the link of `INODE` is 1 (which means that it is the last link), the directory entry and the content of the file are removed, otherwise only the dentry is removed.

All the changes are stored by adding to the log the three new addresses added to the flash store. The flash index remains unchanged.

`DENT` is returned as a negative dentry to specify the deleted file.

The operation *writepage* should also be considered. According to *FSOP#*, *writepage* is executed only if `valid-file (FILE, FS, RI)` (this condition has the same meaning has `valid-file-ino (FILE.ino, FS, RI)`) is satisfiable.

Given a file, a page number and the page to be written, this operation updates the flash store according to the following rules:

- let *INODE* be the inode node corresponding to the input file, if the size of INODE is less or equal than the page number `[ADDRESS, inodenode(inodekey(FILE .ino), INODE . directory, INODE .nlink, PAGENO + 1)]` updates the size of INODE with PAGENO + 1

- `[ADDRESS, datanode(datakey(FILE .ino, PAGENO), PAGE)]` adds a *data node* with the content of PAGE.

Note that instructions in KIV are executed sequentially. When the procedure of *writepage* is analyzed, it is evident that the addresses of both nodes are different.

The RAM Index (respectively journal) is also updated by adding the corresponding keys and addresses (respectively addresses). Once again, the flash index remains unchanged.

3.2.4 Verification

Verification efforts are grouped in three properties: consistency of the file system, functional correctness of the operations and correctness of the replay process.

A fundamental requirement of verifying the file system is its consistency. In order to check if the file system is consistent several predicates were written as, for instance, the predicate *fs-cons*. As said in subsection 3.2.2 *fs-cons* is an invariant of the file system. Therefore, to prove the validity of this statement proof obligations are used. In this case, the formal proof obligation is

fs-cons (fs, ri) $\rightarrow$ *wp(op, fs-cons (fs, ri))*

where *op* refers to any operation of the file system. *wp(op, fs-cons (fs,ri))* denotes the weakest pre-condition of *op* with the respect to a post-condition *fs-cons (fs,ri)*. In other words, *fs-cons (fs, ri)* must be satisfiable before and after the execution of *op*. In KIV, the weakest pre-condition of a program is written using a different syntax. Using the *unlink* operation to replace *op* in a concrete example, the proof obligation for the specified operation is written as follows:

```
fs-cons(FS, RI), valid-dentry(P_INO, DENT, FS, RI), valid-file-ino(DENT .ino,
FS, RI) ⊢ ⟨|unlink(P_INO; DENT, FS, RI, LOG)|⟩fs-cons (FS, RI)
```

The condition above proves that predicates `fs-cons`, `valid-dentry` and `valid-file-ino` must be satisfiable before and after the execution if *unlink*, thus the file system must already be, and also remain, consistent. Note that `valid-dentry` and `valid-file-ino` are included in the proof obligation since they are pre-conditions of the operation.

[36] proved that all the specified operations terminate and carry out post-conditions about their results. Going back to *unlink*, they give and prove total correctness assertions that describe its behavior. The procedure is similar to the procedure for consistency of the file system. Hence, to prove functional correctness the following assertion was used:

```
valid-dentry(P_INO, DENT, FS, RI), valid-file-ino(DENT .ino, FS, RI) ⊢
⟨|unlink(P_INO; DENT, FS, RI, LOG)|⟩valid-negdentry(P_INO, DENT, FS, RI)
```

In order to successfully delete a file, some pre-conditions must be fulfilled. The file identified by `DENT` must be valid and the directory identified by `P_INO` must be valid, containing the file `DENT`. After the deletion, it must be ensured that `DENT` represents a deleted file in the directory `P_INO`, which is ensured by the predicate `valid-negdentry`.

The replay process is represented by the *replay* operation. This, must be able to restore the file system at most the data for the opration been executed when powerlost occurs. Therefore, the predicate *log-cons* was defined to claim that replaying in the current situation will correctly restore the RAM index, meaning that the new RAM index is a copy of the current RAM index. The formal definition that specifies this condition is:

log-cons(fs, ri, fi, log) $\leftrightarrow$ *wp(replay(fs, fi, log; ri2), (fs, ri)* $\cong$ *(fs, ri2))*

A reliable file system should alway preserve the invariant, even in the middle of an operation. The invariant is specified through the following predicates:

log-cons(fs, ri, fi, log) $\wedge$ *store-cons(fs, ri, log)* $\wedge$ *datanode-cons(fs, ri)* $\rightarrow$ *wp(op, log-cons(fs, ri, fi, log)* $\wedge$ *store-cons(fs, ri, log)* $\wedge$ *datanode-cons(fs, ri))*

The definitions of *replay*, *log-cons* and the invariant above applied to the *unlink* operation can be found in Appendix A.1.11, A.1.12 and A.1.13. The invariant above ensures the robustness of the file system when a power loss occurs between operations. Proving the invariance of *log-cons* requires two auxiliary invariants: *store-cons* and *datanode-cons*. *store-cons* is the predicate that requires that each address contained in the RAM index or journal (log) has to be allocated in the flash store.

datanode-cons demands that each datakey of the RAM index belongs to a valid inode and describes a page as the file length. Note that, by transitivity, the same property is applied to all datakeys referred to datanodes of the flash store.

These properties are needed to avoid accessing addresses in the flash store that are not yet allocated, whereas the latter is needed as replaying some operations causes data keys beyond the file size to be deleted.

3.3 Alloy Specification

In this section, the specification written in Alloy will be described. Currently, the model is grouped in several modules: data structures (*ubifs_ds, node, dentry* and *file*), auxiliary functions and operations (*functions*), invariants of the file system (*invariants*), main operations (*operations*) and assertions (*model_checking*). Each kind of module (not necessarily all the models defined) will be discussed in the next.

3.3.1 Data Types

As mention before, data types in Alloy are represented by signatures. Data specifications in KIV are now replaced by signatures. Hence, in the example present in the Section 3.2.1, a main signature is used in the *node* representation. The constructors used in the KIV specification are defined in Alloy as extensions of existing signatures, as presented in Appendix A.2.1. The same method are used to represent keys and it can be found in Appendix A.2.2.

There is a huge difference between KIV and Alloy notation when representing the main data structures of UBIFS. While in KIV there is a generic definition of store which is updated to map nodes to addresses and addresses to keys, in Alloy they are represented using relation multiplicity. Generic specifications and their actualizations are replaced by relations of a certain arity. In the case of flash store, flash index and RAM index, a relation with "arity" 2 is used as depicted in Appendix A.2.3.

The journal can also be easily defined. The set of specifications used in KIV to represent a list, is substituted by sequences. In Alloy, a sequence is a data structure that maps indexes to elements. Its employment can be found in the log definition (Appendix A.2.3).

To finalize the definition of the main data structure of the file system, and as depicted in Appendix A.2.3, a main signature called *UBIFS* containing the relations *fs*, *ri*, *fi* and *log*. All the operations are defined over the UBIFS signature.

The metamodel generated by Alloy Analyzer is in A.2.26.

3.3.2 Functions and Auxiliary Operations

The operations of the UBIFS file system can be classified in two types: basic and main operations. All the functions defined in the *filesystem-base* specification are considered basic operations, and they are defined in Alloy as functions. The module *functions* contains not only the basic operations but also auxiliary functions and predicates used in the Alloy specification. An example of an auxiliary functions in *inodenode_c*. To create an inode node with determined fields in Alloy, the following condition is used:

```
{ one inn : Inodenode | inn·key in _key_ and inn·directory in _directory_ and inn·nlink in _nlink_ and inn·
    size in _size_ }
```

Instead of writing this condition whenever it is necessary to created an inode node, it can be written once in the body of *inodenode_c* (as depicted in Appendix A.2.6). This functions receives as input all the necessary values to return an *Inodenode* with all fields properly filled. In this way (and applying the same procedure to the remaining data types), the definition of the main operations becomes simpler, smaller and more readable.

Recalling the example approached in the Section 3.2.2, here, *getinode* receives as input an Integer (representing the inode number), the flash store and the RAM index and outputs the existing inode in both data structures. However, one must note that the multiplicity factor used in the function (and also some signatures) is *lone* instead of *one*. In the case of *getinode*, the multiplicity factor *one* entails that it always exists an inode according to the specified rules. However, this condition might not be always true (if the functions receives a non-existing inode number). Hence, this means that sometimes *one* can be too restrictive and it might cause some issues in the future. The multiplicity factor *lone* also fulfills all requirements, giving some freedom to the model. Another aspect to consider is the usage of integers instead of natural numbers. Arithmetics is not the strongest feature in Alloy, and the *Natural* data type is still very limited. The module *Integer* provides more arithmetic

operations (for instance, the cardinality of a set) than the module *Natural*. The definition of *getinode* can be found in Appendix A.2.5.

An example of an auxiliary operation is the predicate *new_inodekey*:

```
pred new_inodekey [ink : Inodekey, ri : Key →lone Address, fs : Address →lone Node]
```

This predicate is used in the operations *create* and *mkdir* and it ensures that `ink` is an absent key in the flash store and RAM index. In KIV, this condition is directly written in the body of the operation. However, in Alloy, special attention must be payed to this condition. Further details will be presented in Section 3.3.5.

3.3.3 Invariants

In Alloy, axioms are translated using predicates. In KIV, the specification *filesystem-base* contains most of pre and post-conditions and most of invariants, in Alloy, they are grouped in the module *invariants*. All predicates were interpreted as invariants (note that the same predicate can work as an invariant or as a pre-condition). Invariants were classified in two categories: referential integrity and file system consistency. Referential integrity ensures the references between data structures are valid. An example of a predicate that ensures referential integrity is *valid-ino* that given a flash store, a RAM index an an inode number, ensures that the inode number in the range of RI and FS is valid:

- the inode number must be bigger than zero;
- at least, an inodekey should exist such:

  it is identified by the given inode number;

  it exists in the RAM index;

  its corresponding address exists in the flash store.

In KIV, these conditions are enough to define *valid-ino* as depicted in Appendix A.1.9, . However, in Alloy, this definition creates some redundancy. When instantiating the predicate, Alloy Analyzer produces the output in 3.3.3.

Carefully analyzing the output mentioned before, it is possible to find some inconsistencies: `UBIFS` is the file system consisting of 4 relations: `log` (containing `Address2` in the first position), `fi` (the flash index), `fs` (that contains the relations `Address0 -> Datanode` and `Address2 -> Datanode`) and `ri` (containg the relations `Datakey0 -> Address0` and `Inodekey -> Address0`). `Inodekey` is the key identified by the inode number `7` and it is mapped in the RAM index by `Address0`. So far, `Inodekey` conforms to the conditions specified in the predicate. The last condition assures that `Address0` must be stored in the flash store. According to the relation `fs`, the condition holds once `Address0` maps `Datanode`. Although the predicate holds, two inconsistencies can be found. First of all, in the RAM index, the same address is mapped by two different keys and obviously this situation can not occur.

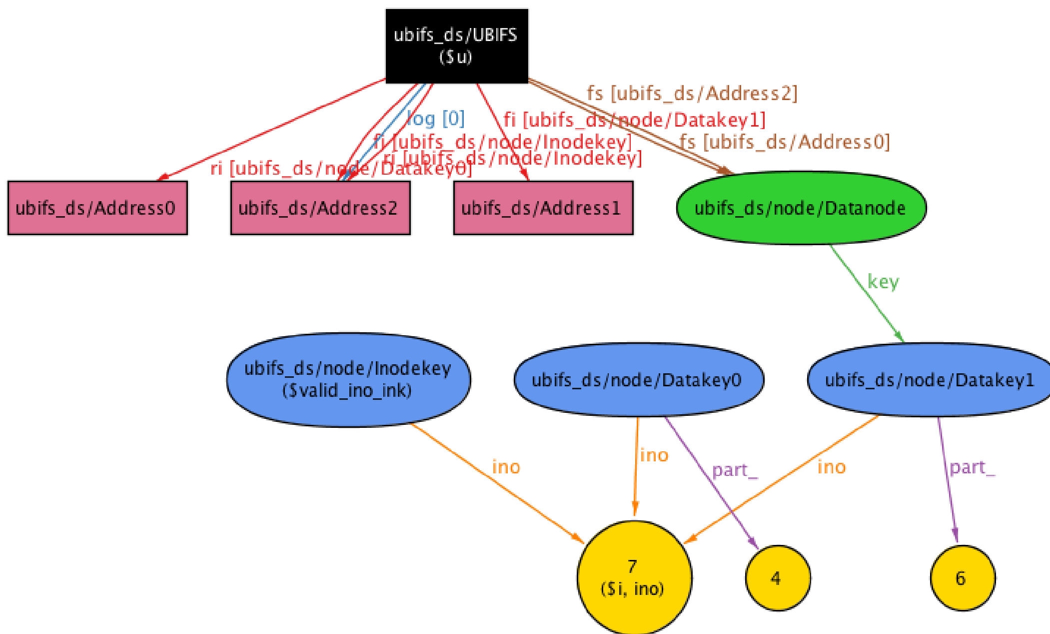


Figure 3.1: Instance of valid_ino produced by Alloy Analyzer

The second inconsistency involves the atoms `Address0`, `Inodekey` and `Datanode`. According to the structure of the UBIFS, the RAM index is used to quickly access nodes with a given key. This presupposes that the keys mapped by a given address belong to the node pointed by the same address, in the flash store. However, in the instance generated by Alloy Analyzer, RAM index maps `Inodekey` to `Address0` and `Address0` mapps `Datanode` in the flash store. Obviously, not the of `Datanode` is `Inodekey` neither an inodekey can be associated to a datanode.

A similar situation could also occur (and it was checked during the verification). By default, Alloy assumes that it exists more than a key with the same inode number, i.e., two different keys share the same inode number. Considering an inodekey mapped to a certain address, the flash store maps the same address to a certain inode node (refered by the inodekey stored innthe RAM index). however, since only inode numbers are compared (instead of keys) to ensure the consistency of the keys and inodes, it is possible the inodekey in the RAM index being different from the key that refers the inode node in the flash store.This kind of situations occur because this condition only ensures that the mapped address in the RAM index is in the range of the flash store (it is not specified any relation between the keys in the RAM index and the nodes in the flash store).

To avoid this kind of issues, another condition was added to the definition of *valid-ino*: the association between the stored in the RAM index and the key of the corresponding node in the flash store. This way, all the problems are solved, avoiding some counter-examples in the verification. The full implementation in Alloy of *valid-ino* is displayed in Appendix A.2.8.

Note that in the definition of the predicate, the implication was used. The main goal is to reduce the range of input atoms, i.e, only the keys stored in the RAM index are included.

File system consistency includes the main invariants of the file system. Using the example of Appendix 3.2.2, *fs-cons* is an invariant that ensures consistency of the file system. In Alloy, it has

the same signature as in KIV. However, *fs-cons* is defined using the predicate *fs-key-cons* and here dwells the main difference of both definitions. In KIV, an excerpt of the condition that tests the type of the inputed key is:

`key .dentry?` $\rightarrow$ `fs-dentry-cons(key, fs, ri)`

In Alloy, it is ensured by the condition:

```
{key_ in Dentrykey ⇒ fs_dentry_cons[key_, fs, ri]}
```

Again, special attention must be payed to the Alloy condition. The operator = was replaced by the operator *in* to avoid the condition been too restrictive. The operator = specifies that the relation `key_` is the same as the relation `Dentrykey`, hence the predicate *fs_dentry_cons* would only be applied to one and only one key, instead of being applied to all dentrykeys. Therefore, replacing = by *in* specifies that `key_` contains `Dentrykey`. The full definition of *fs-key-cons* can be seen in Appendix A.1.9 for the KIV definition and in Appendix A.2.9 for the Alloy definition.

fs-key-cons is a quite complex invariant. As mentioned before, it has three variants, according to the existing kinds of keys. As the name suggests, *fs-dentry-keys* checks the consistency of dentrykeys, according to the following conditions:

- there is an inode node in the flash store identified by the corresponding inodekey (the inodekey and the dentrykey have the same inode number) that refers a valid directory;
- the dentrykey and the key of the dentry node transitively mapped by it, have the same name;
- the dentry node transitively mapped by the dentrykey must correspond to a valid inode node (by its the inode number);

The specification in KIV is written using the following axioms:

- valid-dir-ino(key.ino, fs, ri)
- key.name = fs[ri[key]].name
- valid-ino(fs[ri[key]].ino, fs, ri)

The axiom *valid-dir-ino* uses the function *getinode* to select the inode node identified by `key.ino`. However, the flash store may store several inode nodes with the same inode number with different properties and this genereted some counter-examples during the verification. In order to weaken the invariant, the predicate `valid-dir-ino` was replaced by the conditions of its body but with a difference: the multiplicity factor *all* was replaced by the multiplicity factor *some*, ensuring that only some inode nodes hold the predicate. Further details of this issue can be found in Section 3.3.5.

The definition of *datanode-cons*, *nodekey-cons* in Alloy is similar to their definition in KIV. Both definitions can be found in A.2.13 and A.2.14.

The predicate *log-cons* was defined using point-free notation: stating that

($\forall$ adr. adr $\in$ log $\vee$ adr $\in$ ri $\rightarrow$ adr $\in$ fs);

is the same as

```
(log·rng + ri·rng) in fs·dom
```

Sequences in Alloy are just relations where an index is maped to an element. In this particular case, a sequence of addresses is a relation that maps indexes to addresses (`Int -> Address`), hence the indexes represent the domain of the relation and the set of the addresses stored in the log belong to the range of the relation.

The operator $\vee$ was replaced by the operation "union" in Alloy to specify that the union between range of the log and the range of the RAM index is a subset of the domain of the flash store. This way, the predicate *store-cons* became simpler and easier to read.

3.3.4 Operations

The module *operations* contains all the operations of the file system. Due the fact that operations modify the state of the file system, they are interpreted as predicates. In both specifications, operations are defined as predicates, however there is a big difference between both definitions. The first main difference refers to the existence of a top-level operation, *FSOP#* that selects one of the available operations, that establishes the necessary pre-conditions of each operation. In Alloy, pre-conditions are introduced directly in the corresponding operation. Hence, the definition of a top-level operation is no longer needed.

Another main difference in both definitions is the signature of the operations. Continuing with the example of *unlink*, its signature in Alloy contains, as arguments, all the needed inputs and also the all the outputs of the operation. Note that predicates do not have a return state. Here, output means the modified state of the involved variables. In KIV, the main variables involved in the definition of the file system are declared globally and, for each operation, only the modified variables are called. In Alloy, all the entire file system is an argument. This property is showed in the definition of *unlink* in A.2.15.

Transcribing the signature of the operation, one has:

```
pred UNLINK [p_ino : Int, dent : Dentry, dent' : Negdentry, u, u' : UBIFS]
```

where `p_ino`, `dent`, `u` are input arguments. There are not global declarations of variables in Alloy and, in a certain data type, it must be specified not only the modified fields by the operation but also all that remain unchanged (friend condition). These are the four reasons to *unlink* receiving the entire file system as input.

The body of the operation is divided in pre and post-conditions. There are two definitions (using two axioms) of the predicate `valid-dentry` in KIV. Despite the existence of two different axioms, only a predicate is defined. Alloy Analyzer does not allow duplicated predicates,

hence, in the current example, the predicate `valid-dentry` in KIV was replaced by the predicate `valid_dentry_with_parent` (it received the same name as the corresponding axiom).

The post-conditions encompass the procedures of the operation and the updated involved data types. In this operation, `dent'` is the returned negdentry and `u'` is the final state of the file system.

In order to assure that `adr1`, `adr2` and `adr3` are new addresses in the flash store, the axiom *new* was used in KIV. In Alloy, this axiom was replaced by the following conditions:

- **one disj** adr1, adr2, adr3 : Address – ensuring that the three addresses are disjoint. When nothing is specified, Alloy Analyzer can assume that, for instance, adr1 and adr3 are equals and it could generate some inconsistencies (two different nodes are allocated in the same address).
- (adr1 + adr2 + adr3) **not in** dom[u.fs] – specifying that this is a new subset of addresses in the flash store.

As said before, Alloy Analyzer as a limited state space called scope. When running an instance under a certain scope, Alloy Analyzer generates as many atoms as scope allows, sometimes unnecessary atoms. In a big model such as UBIFS, Alloy Analyzer might not have enough scope to create the instance. A solution is to increase the scope, increasing the scope, however, entails exponential increase in solving time. Another solution is trying to reduce the number of created atoms that can be done using the instruction *let*. The *let* statement acts as a macro replacing the right-side of the assignment by the left-side of the assignment, i.e. it creates temporary three new variables to be used to update the file system in the boundary of the *let* clause.

Another advantage of the *let* statement is avoiding the issues caused by repeating the use of the constructors. Focusing on the following excerpt of *unlink*:

```
...
in1 = inodenode_c[inodekey_c[p_ino], p_inode·directory, p_inode·nlink, inode·size−1],
  {
    u'·fs = u·fs + (adr2 →in1)
    ...
    ri_temp = u·ri − temp + (in1·key →adr2)
    ...
```

it is easy to conclude that the node `in1` is added to the flash store and its key is added to the RAM index. Repeating the constructor `inodekey_c[p_ino]` can cause two situations: one key is created and the key of the inode node is the same as the key stored in the RAM index or two different keys with the same inode number and the key of the created inode node is different from the key stored in the RAM index. Apparently it could not raise any issues, however, when checking the consistency of the keys, Alloy Analyzer will generate a counter-example: having the same inode number does not presuppose being the same key.

The definition of *unlink* has another difference from the implementation in KIV. Besides marking the file as removed in the flash store, deleting a file or directory also involves removing its dentrykey

from the RAM index. At first sight, this procedure could be literally done and in fact, this is what happens in KIV. The first implementation in Alloy also contained a statement that was the literal translation of KIV. That statement was replaced by the function *dentrykeys_name* that returns the set of the existing dentrykeys in the RAM index, according to certain parameters. Hence, let `dkns` be the set of dentrykeys, as follows:

```
dnks = dentrykeys_name [u·ri, p_ino, dent·name]
```

The RAM index is a data structure that maps keys to addresses. Hence, to remove `dkns` from the RAM index it is necessary to create a relation mapping the keys from `dkns` to their corresponding addresses which is represented by the following statement:

```
temp = (dnks <:u·ri)
u'·ri = u·ri − temp
```

The operator *<:* restricts the domain of a relation, hence `temp` contains the relations mapping all the keys from `dkns` to addresses. Now, removing the intended keys from the RAM index is done using the traditional way.

The *let* clause was also used to update the log, but for a different purpose. Operations under sequences in Alloy requires using an auxiliary module. Given a sequence and an element, the operation *add* selects the biggest index of the sequence to attribute a new index to the added element, meaning that only an element can be added at once.

As opposed to the KIV specification, in Alloy, it is essential to specify that the flash index remains unchanged as can be seen in the full definition of the operation.

The operation *writepage* also has an interesting detail: the instruction *if-then-else*. In Alloy, ***if*** *a* ***then*** *b* ***else*** *c* it represent by the statement *a* ***=>*** *b* ***else*** *c*. Once the update of the flash store and RAM index depends on the page number and the size of the inode node refered by the input file, the entire file system is updated in the boundary of the *if* statement. The full definition of the operation can be found in Appendix A.2.16.

This operation has yet another curious detail related to the invariant *datanode-cons*. Further details can be found in Section 3.3.5.Figure 3.3.4 illustrates the instance generated by Alloy Analyzer for the current operation.

In order to clarify the changes between `u` (the atom `UBIFS1`) and `u'` (the atom `UBIFS0`), the Alloy evaluator was used. Figure 3.3.4 (respectively Figure 3.3.4) illustrates the flash store and RAM index of `u` (respectively `u'`).

3.3.5 Verification

Another great difference between KIV and Alloy is the way verification is done. KIV, a theorem prover, as the name suggests, tries to prove that a theorem is correct. When a proof fails, it can be hard to find what cause the failure: whether the theorem is invalid, or whether the proof strategy failed.

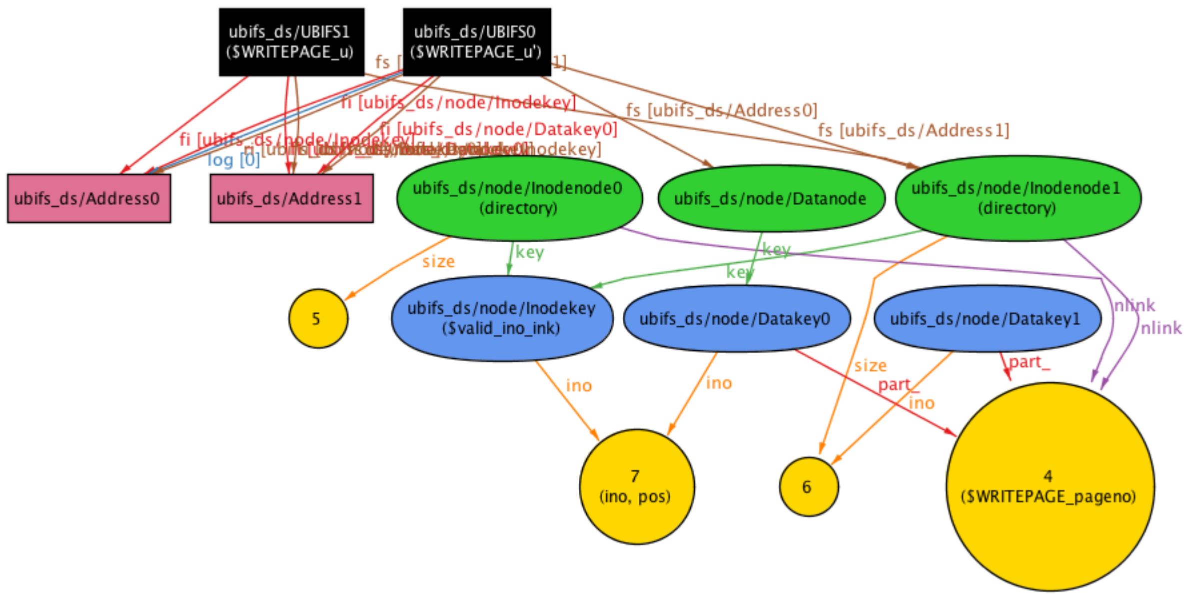


Figure 3.2: Instance of the operation writepage

ubifs_ds/UBIFS$1.fs

{ubifs_ds/Address$1->ubifs_ds/node/Inodenode$
1}

ubifs_ds/UBIFS$1.ri

{ubifs_ds/node/Inodekey$0->ubifs_ds/Address$1
}

Figure 3.3: Flash store and RAM index
of u

ubifs_ds/UBIFS$0.fs

{ubifs_ds/Address$0->ubifs_ds/node/Datanode$0,

ubifs_ds/Address$1->ubifs_ds/node/Inodenode$1
}

ubifs_ds/UBIFS$0.ri

{ubifs_ds/node/Datakey$0->ubifs_ds/Address$0,
ubifs_ds/node/Inodekey$0->ubifs_ds/Address$1}

Figure 3.4: Flash store and RAM index of u'

A model-checker, such as Alloy Analyzer, tries to find counter-examples to certain assertion. It works as a "refuter" rather a "prover". If a counter-example is generetad, the assertion is invalid, however if no counter-example is found, the assertion may be valid or may still be invalid. Increasing the scope of the solver will reduce the probability of the assertion being invalid.

In Alloy, verification efforts followed the KIV rules and they are also divided into 3 groups: consistency of the file system, functional correctness of the operations and correctness of the replay process. The theorem base for the *filesystem-asm* specification[1] was the guideline to check the consistency of the file system and functional correctness of the operations. Most of the axiom proved in the theorem base were reproduced in Alloy as assertions. Starting by "auxiliary" axioms (axioms used by other axioms, such as main invariants), *dir-two-links* is an axiom used by the invariant *fs-cons* in the operation *rename*. In Alloy, these axioms were not used in any invariant, however its corresponding assertion was defined and checked for counter-examples.

Verifying the UBIFS file system in Alloy Analyzer took quite some time (the entire file system

[1] `http://www.informatik.uni-augsburg.de/swt/projects/Flash/UBIFS/specs/filesystem-asm/export/lemmasummary.xml`

took a few hours), particularly from the SAT solver.By default Alloy Analyzer uses *SAT4J* as solver which is enough for small models. UBIFS is a large model and SAT4J was insufficient (for instance, the assertion *fs-cons-same-key* (which is a small assertion) took 3 minutes to be solved!). For large problems, it is recommended to *Berkmin* solver. However, due the incompatibility between the solver and the operating system (Berkmin is only available for the Linux and Solaris platform and the operating system was Mac Os X), Berkmin was replaced by *minisat*. According to Alloy FAQ[2], minisat is a good choice for small problems. However, minisat had a better performance than SAT4J and although the solving time of certain assertions continued to be considerably large, it has proved sufficient for the problem.

As said before, the consistency of the file system includes checking, for each operation, the preservation of four main invariants: *nodekey-cons*, *store-cons*, *datanode-cons* and *fs-cons*. Following the theorem base, an assertion was created combining an operation with an invariant (for a total of 55 assertions). The invariant *nodekey-cons* did not raise any major problems (probably due the *let* statements used in the operations). An example of this invariant applied to the *unlink* operation (assertion *nodekey_cons_unlink*) can be found in A.2.19.

Sometimes, point-free notation can be too restrictive. Considering a relation as an example, specifying that only some elements (instead of all) of the relation hold a given condition in PF notation might not be trivial. In this case, PW notation woul be an advantageous option. However, although *store-cons* being defined using point-free notation, no counter-examples were found in any operation. The verification of *store-cons* is similar to *nodekey-cons*: an assertion was defined to each operation.

The operation *writepage* raise some difficulties when checking for the invariant *datanode-cons*. In KIV, the corresponding theorem (name *datanode-cons-writepage*) stated that the conditions *datanode-cons* and *valid-file-ino* must hold before the execution of *writepage*, and after its execution *datanode-cons* must continue hold. In Alloy, the weakest pre-condition specified in KIV is replaced by an implication. The corresponding assertion in Alloy is written as follows:

```
all u, u' : UBIFS, file : File, pageno : Int, page : Page |
    valid_file_ino [file·ino, u·fs, u·ri] and
    datanode_cons [u·fs, u·ri] and
    WRITEPAGE [file, pageno, page, u, u'] ⇒ datanode_cons [u'·fs, u'·ri]
```

As in other operations, the variable u represents the previous state of the file system and the variable *u'* represents the final state of the file system.

Note that using the multiplicity factor *all* entails that the conditions are held for every instance generated by Alloy Analyzer. If another multiplicity factor was used, for instance, the multiplicity factor *some*, the conditions were held only for some file systems, meaning that it could have at least, one file system where the conditions were not valid, producing an inconsistent file system. However, when checking this assertion, Alloy Analyzer found a counter-example, illustrated by Figure 3.3.5.

[2] http://alloy.mit.edu/faq.php

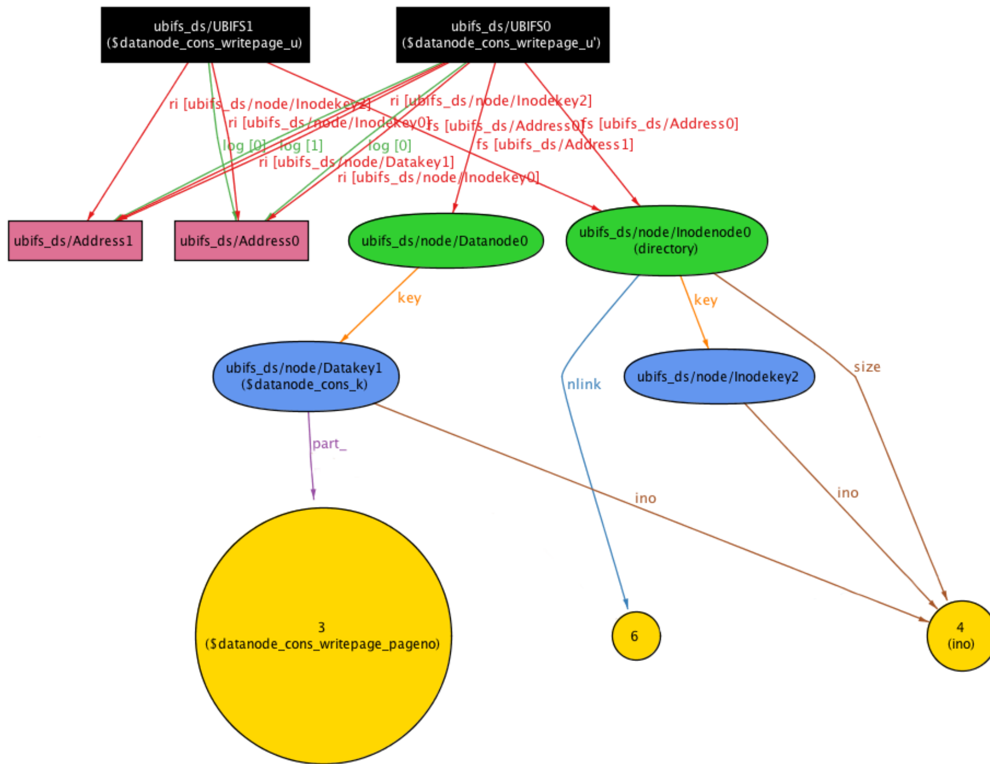


Figure 3.5: counter-example of datanode_cons_writepage

Before analyzing the counter-example, the new nodes adds to the flash store (and corresponding keys to the RAM index) must be identified. This task can be easily done using the *Evaluator*. Figures 3.3.5 and 3.3.5 illustrate the flash store the RAM index from both states of the file system.

ubifs_ds/UBIFS$1.fs

{ubifs_ds/Address$0->ubifs_ds/node/Inodenode$0}

ubifs_ds/UBIFS$1.ri

{ubifs_ds/node/Inodekey$0->ubifs_ds/Address$0, ubifs_ds/node/Inodekey$2->ubifs_ds/Address$1}

Figure 3.6: Flash store and RAM index of u

ubifs_ds/UBIFS$0.fs

{ubifs_ds/Address$0->ubifs_ds/node/Inodenode$0, ubifs_ds/Address$1->ubifs_ds/node/Datanode$0}

ubifs_ds/UBIFS$0.ri

{ubifs_ds/node/Datakey$1->ubifs_ds/Address$1, ubifs_ds/node/Inodekey$0->ubifs_ds/Address$0, ubifs_ds/node/Inodekey$2->ubifs_ds/Address$1}

Figure 3.7: Flash store and RAM index of u'

Only `Datanode` (respectively `Datakey`) was added to the flash store (respectively RAM index), meaning that, according by the behavior of the operation, the page number is smaller than the size of the inode node refered to the input file. And the RAM index of the final state has one inconsistency: the same address maps two different keys, but does it really interfere with the invariant violation?

In the counter-example, Alloy Analyzer identifies the signatures that breakes the invariant, hence `pageno` has value 3 which is also the value of the `Datakey.part_`. 4 is the value of the inode number that identifies `Datakey`, `Inodekey0` and `Inodekey1`. Apparently, the invariant would not

getinode[ubifs_ds/node/Datakey$1.ino,
ubifs_ds/UBIFS$0.fs, ubifs_ds/UBIFS$0.ri]

Figure 3.8: Output of getinode under the counter-example of 3.3.5

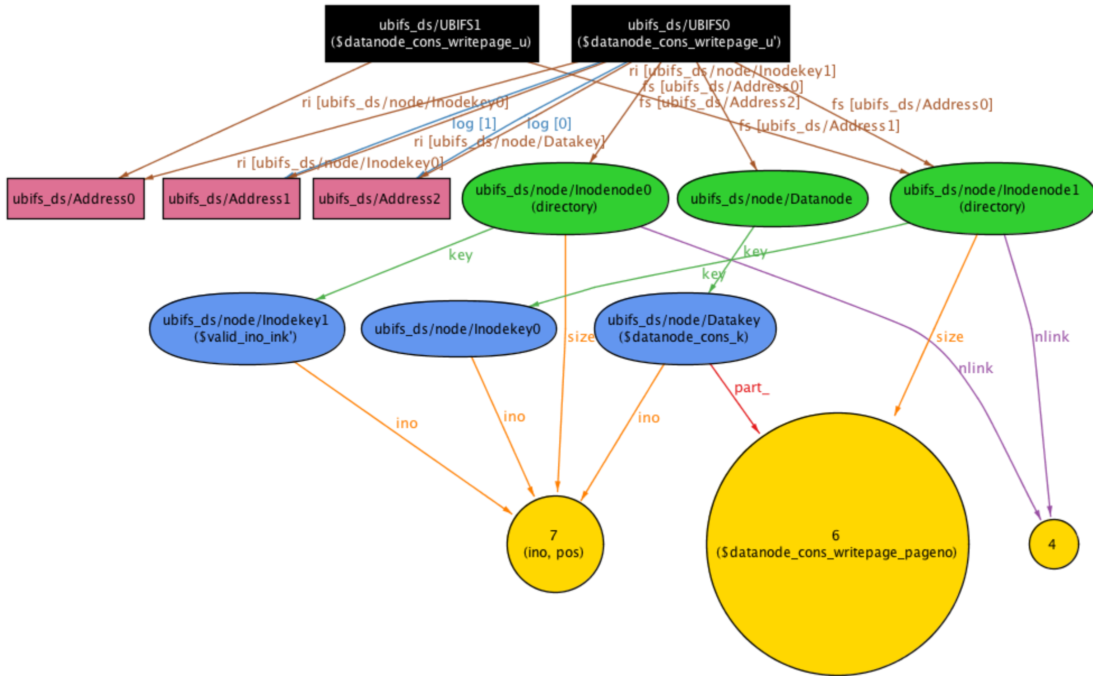


Figure 3.9: counter-example of datanode_cons_writepage

be broken, since the size of both inode nodes is bigger than the part size of the datakey. So, what causes the counter-example? In fact, the inconsistency of the RAM index really influenced the invariant through the function *getinode*. According to the definition of *getinode* and the inconsistency of the RAM index, there is not any inodekey that holds the conditions of the function, causing the output illustrated by Figure 3.3.5.

The solution involves ensuring the consistency of the data structures by adding the predicate (and also invariant) *store-cons* to the assertion. This way, the problem of the result of *getinode* will be solved. However, will these conditions be sufficient to ensure the preservation of the invariant? The answer is no. After the addition of *store-cons*, Alloy Analyzer found another counter-example, depicted in Figure 3.3.5.

In this case, `Inodenode1` is the existing inode in the flash store both `Inodenode0` and `Datanode` are the new added nodes that, according to the behavior of the operation, mean that the the page number is smaller than the size of the inode node refered by the input file. In fact, the inode inode is updated with the new size (pageno + 1) but the old inode node still remains in the flash store. When the predicate *datanode_cons* is ran for the updated file system, the new inserted datakey will be compared to old inode node instead of the updated inode node. In other words, the part number of `Datakey` is compared to the size of `Inodenode1` when it should be compared to the size

of `Inodenode0`.

The solution is to check the properties of the predicate only for the added nodes and keys. The existing data is checked and it remains unchanged meaning that it remains consistent. Hence, the full definition of *datanode_cons_writepage* can be found in Appendix A.2.20.

However, according to the structure and behavior of UBIFS, it does not make any sense adding a new data node without adding the corresponding inode node. This suggests that the definition of the operation *writepage* is wrong: the flash store and RAM index are only updated when the size of the chosen inode node is smaller than the inputed page number. Thus, the right definition of *writepage* can be found in Appendix A.2.17.

Most of the counter-examples found are related, particularly to operation *writepage*. The exception is for the use of *store-cons*. This predicate was added to all assertions "datanode_cons". In some operations no counter-example was found but it could generate some inconsistencies, hence, to avoid possible issues, the predicate is used to ensure the consistency of the file system.

The invariant *fs-key-cons* also presented many issues in most operations. First of all, *fs-key-cons* is a complex invariant. As said before, it has three variants but only one of the three variants presented some inconsistencies for several reasons. The predicate *fs-data-cons* is very simple and it did not bring many issues. Actually, *fs-data-cons* only ensures that the inode number of the data key regards a valid file as follows:

```
pred fs_data_cons [key_ : Datakey, fs : Address →lone Node, ri : Key →lone Address]
{
  valid_file_ino[key_·ino, fs, ri]
}
```

However, due to the conditions of the predicate *valid_file_ino*, this predicate is only ensured for the new added inodes and keys. The issues caused by this predicate are similar to the issues caused by *datanode_cons_writepage*.

The predicate *fs-dentry-cons* (available in Appendix A.2.10) is a little more complex than *fs-data-cons* but it did not bring many issues either. Again, similarly to *fs-data-cons*, it is only ensured for the new added inodes and keys. However, a curious case emerged. Validating a dentrykey entails ensuring the following conditions:

- the inode number of the inputed key must refer a valid directory
- the name of the key is consistent with the name of the corresponding dentry node
- let *dentrynode* be the dentry node identified by the inputed key, such as *dentrynode* must refer a valid hard link (the field `.dino` must contain a valid inode number)

Apparently, there was not any reason for Alloy Analyzer to find a counter-example. The first condition is ensured by the predicate *valid_dir_ino* that establishes the following conditions:

- the inputed inode number is valid inode;
- the corresponding inode node refers to a directory (i.e., the field `.directory` is True);
- the corresponding inode node refers to more than one hard link (i.e., the field `.nlink` is bigger than 1).

In the assertion *fscons_create*, all this conditions holds except the last one. Apparently, the pre-condition of the operation *create* is not sufficient to ensure the validity of a directory. The same predicate is used to ensure the pre-condition and to ensure the validity of a dentry key. However, only the third condition of *valid_dir_ino* is violated. There no reason that explains this occurence, and the only solution found to solve this problem is adding the same predicate as a post-condition. The full definition of *create* can be seen in Appendix A.2.18.

The big problem of *fs-cons* is the predicate *fs-inode-cons*. This predicate comes also in two variants: *fs-file-cons* for inode nodes representing files and *fs-dir-cons* for inode nodes representing directories. For all inodekeys of the file system, the predicate *fs_file_cons* (available in Appendix A.2.11) ensures the following conditions:

- let *inodenode* be the inode node identified by *inodekey* and *links* the set of dentrykeys that identify all dentrynodes working as hard links of the *inodenode* such as the numbers of links of *inodenode* is the cardinality of *links*;
- the part number of all the existing datakeys in the RAM index related to the inputed inodekey, must be smaller than the the size of the corresponding inode node.

Similarly to *datanode_cons_writepage*, duplicated inodes also caused some issues: on the one hand not all inode nodes of the updated file store have the numbers of links updated (to ensure the first condition). On the other hand, not all inode nodes have the updated size (to ensure the second condition). The solution was also to ensure the conditions only for the new added nodes (respectively keys), as follows:

```
...
let fs = u'·fs − u·fs, ri = u'·ri − u·ri |
  {
    all n : Node | n in fs·rng and n·key in ri·dom ⇒
    {
        n in Inodenode ⇒ fs_inode_cons [n·key, fs, ri]
        ...
    }
```

This solution does not solve the problem. In fact, the first condition of the predicate is also broken. The new added inode node contains the updated number of links (i.e. updated field `.nlink`), however the number of hard links (provided by the function *links*) of the updated flash store (and

RAM index) is not the same as the number of hard links in the mapping `fs`, hence the conditions `fs_inode_cons [n.key, fs, ri]` and `fs_inode_cons [n.key, u'.fs, u'.ri]` can not be used. The solution is even more robust: defining an auxiliary *fs_inode_cons_aux* that matches, whenever necessary, the new added nodes with the updated flash store. Thus, the first condition of the invariant was replaced by the following condition:

```
...
some inn : Inodenode | inn·key in key_ and inn in fs·rng and key_ in ri·dom ⇒
{
    inn·nlink in #links[key_·ino, fs', ri']
    all key2 : Datakey | key2 in datakeys[ri, key_·ino] ⇒ key2·part_ < inn·size
}
```

Note that `fs'` and `ri'` are the updated flash store and RAM index and `fs` and `ri` are the relations containing only the inodes and keys recently added.

The same procedure was applied to the invariant *fs_dir_cons*, due its similarity with *fs_file_cons*. The full definition of *fs_dir_cons* can be found in Appendix A.2.12

The idea of functional correctness in Alloy is similar to KIV: some preconditions must hold before the operation and, some postconditions must hold after the operation. In KIV, the theorems that prove functional correctness are identified by *prepost-op*. In Alloy, the same name was maintained. Two examples of prepost assertions are *prepost_unlink* and *prepost_writepage*.

Checking functional correctness also had some issues. In fact, most of additional conditions in the operations (see subsection 3.3.4) are added to avoid counter-examples related to the functional correctness. Nevertheless, there still are some conditions that should be emphasized. Starting by *prepost_unlink* and focusing on the following excerpt of *unlink* operation:

```
dent' = negdentry_c[dent·name]
...
let dn1 = dentrynode_c[dentrykey_c[p_ino, dent'·name], dent'·name, 0],
```

where `dent` is the inputed directory entry and `dent'` is the returned directory entry, it is obvious that both dentries have the same name. Theoretically, in the constructor of `dn1`, it could be used `dent.name` instead of `dent'.name` but, in fact, it could cause some inconsistencies. Actually, Alloy Analyzer found a counter-example for the assertion. The fact that `dent` and `dent'` sharing the same attribute does not entail being the same atom. Moreover, they are different atoms (`dent` is a dentry and `dent'` is a negative dentry). Replacing `dent'.name` by `dent.name` presupposes that there is not any connection between `dent'` and the created nodes (and the final state of the file system). If there is not any connection to the returned file system it is impossible to ensure that `dent'` is a valid negative dentry in the returned file system (the predicate *valid_negdentry*).

The assertion *prepost_writepage* has two interesting details. First of all, there is not any theorem in KIV with the same name. The corresponding theorem is called *prepost-writereadpage*. According to [36], given postconditions for *readpage* and *writepage* individually turned out to be rather hard when trying to remain implementation independent, so the authors decided to use the combined

postcondition that reading data after writing returns exactly the data written. In Alloy the combined theorem was replaced by two separated assertions: *prepost_readpage* and *prepost_writepage* that can be found respectively in Appendix A.2.22 and A.2.23.

There are several conflicts between old data and updated data, in most assertions. Sometimes the invariants (or conditions) are too weak and additional conditions are needed. Sometimes the invariants are too strong causing conflicts between old and updated data. An example of this situation is *prepost_writepage*. Similar to *datanode_cons_writepage*, the postcondition was only ensured for the new added data by similar reasons.

The operation *lookup* presented additional issues. In the UBIFS there is an inode node refered by a dentry node, i.e., the inode number of a dentry node (the field `.dino`) always refers to an inode node. In KIV, nothing is specified in the operation *lookup* because it was previously assumed. However, in Alloy, this condition must be specified in the operation, otherwise it is not possible to return a valid dentry (i.e., Alloy Analyzer shows a counter-example where there is not any inode node in the file system refered by the returned dentry.) In order to avoid this situation, the following conditions were added:

```
{ (dk in u·ri·dom) ⇒ some dn : Dentrynode |
        {valid_ino [dn·dino, u·fs, u·ri] and dn·key in dk and dent' = mkdentry_c[dent·name, dn·dino] }
  else dent' = negdentry_c [dent·name]
}
```

This condition is enough to ensure the validity of the returned dentry. The body of the operation is similar to its definition in KIV, hence, for this reason, the full definition in Alloy is not presented. The only difference is presented in the excerpt above.

Checking the correctness of the replay process also have some particularities. The problems arose not in the *replay* operation itself but on the underlying predicates. Replaying a file system entails setting it to an isomorphic state.

Given two file systems, the predicate *iso* ensures that they are isomorphic through the predicate *key-iso*. Let *ri* (respectively *fs*) and *ri2* (respectively *fs2*) be the RAM and flash store of two file systems such as they are isomorphic if:

- all keys stored in the RAM index of both file systems are the same ($key \in ri \leftrightarrow key \in ri2$)
- for all keys stored in the RAM index, the mapped addresses must be stored in both flash store ($key \in ri2 \leftrightarrow ri[key] \in fs \land ri2[key] \in fs2$)
- the nodes of the flash stored are mapped in the same position ($fs[ri[key]] = fs2[ri2[key]]$)

Being isomorphic does not mean being exactly identical. In KIV, the *replay operation* and the predicate *iso* assure the isomorphic definition. In Alloy, the definition of *log-cons* is identical to KIV, however the predicate *iso*, particularly the predicate *key_iso* raise some inconsistencies. Saying the second condition above does not presuppose that the same address is mapped and consequently

the third condition could also failed. After several attempts, the only solution found is to ensure that both flash stores and both RAM index are identical:

```
pred iso [fs : Address →lone Node, ri : Key →lone Address, fs2 : Address →lone Node, ri2 : Key →lone
     Address]
{
  fs = fs2
  ri = ri2
}
```

This way, the predicate *key_iso* defined in Alloy is not used. It might be too strong but it was the solution found to avoid counter-examples in the replay process. The full definition of *log_cons* (for instance, to the operation *unlink*) can be found in Appendix A.2.24.

As said before, integers and natural numbers are limited data types. Natural numbers were the first choice to represent the natural numbers in KIV. However, operations under sets (such as cardinality, that was used in several *fs-cons* invariants) returns integers, hence the data type *Natural* was replaced by the data type *Int*. In Alloy, integers also contain negative numbers meaning that the cardinality of a set could be a negative number and negative numbers could cause some issues in some invariants, such as *datanode-cons* and *fs-cons*. Hence, another constraint must be added to the file system:

- the size and the number of inode nodes must be a non-negative number;
- the number of related nodes of dentry nodes (field *.dino*) must be a non-negative number;
- the part number of datakeys can not be a negative number.

To perform these constraints, a fact was added in the data structures of nodes and keys (*node.als*) named *Notnegative* and it can be found in Appendix A.2.25. In KIV, inode numbers of the keys are also natural numbers. However, inode number are mere identifiers hence there is no reason to specify that they should be positive or negative numbers. Actually, there is no reason to inode numbers to be integers (there is not any operation under inode numbers), however, to keep the model in Alloy close as possible to the abstract specification in KIV, the data type *Integer* was kept.

Overall, there is a small difference between the invariants on the rename operation. In KIV, sometimes there are two theorems for the same invariants. Initially, in Alloy, the theorems were kept but, after further analysis, it was decided to match the two theorems in one assertion.

Because it is declarative, in Alloy it does not make any sense proving the termination of an operation. Hence, all the theorems concerning the termination of an operation (theorems *term-op*) were ignored.

Chapter 4

Concluding Remarks and Future Work

This chapter is devoted to concluding from the work presented in this dissertation and pointing to opportunities for future reserach: which lessons have been learnt, what the major difficulties were, which goals were not achieved and lead to work that can still be done. This chapter is split in two sections. Section 4.1 will describe the goals achieved and review advantages and disadvantages of each tool, including difficulties that arose during the development of the project."

Section 4.2 points to future work: the goals not achieved and the opportunities for enhancing and extending what was achieved.

4.1 Concluding Remarks

4.1.1 Contributions

A model was built specifying the structure and behavior of a journaled file system, particularly the UBIFS for flash memories. Following the abstract specification available in the KIV projects website [1] and following the guidelines of [36], the abstract model was build fulfilling its main goal: submitting the specification proved by a theorem prover to a model-checker and doing a comparative work n respect of a number of criteria as aspects described in Section 1.4.

The Alloy model of the UBIFS is complete and no counter-example was found, thus authenticating the robustness of the model. The specification developed in Alloy intentionally contains several similarities to the specification in KIV, the main idea was making easier the comparison between both languages.

Data structures

The specification in Alloy keeps several data structures of the file system created in KIV. However, the full data structure of the model in Alloy became smaller and simpler: the number of signatures defined is much smaller than the number of specification defined in KIV as seen in the graphical overview of each model). The map relations used in Alloy are also another advantage: defining the

[1] `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`

flash store, RAM index and flash index using map relations made everything smaller and simpler than using generic specifications and actualizations. The definition of the journal was also advantageous due to the type of data provided by Alloy. In fact, integers (and natural numbers) were the only disadvantageous data types due to their limitation (the main goal of Alloy is building an abstract model and integers are a concrete data type). The main advantage of data types in KIV are the natural numbers. Once they do not have any limitations, they can be used without any problems, unlike Alloy.

Invariants and auxiliary operations

Invariants and auxiliary operations in Alloy were directly translated from KIV. Hence their complexity in Alloy is similar to the complexity in KIV, even with the additional changes due to model-checking, i.e, the specification in Alloy is similar to the specification in KIV. This turned out to be an advantage: the translation of KIV for Alloy was easily performed. The only inconvenience was to understand the meaning of some operations over keys and nodes (particularly in some functions such as *dentrykeys* and *links*): some confusion arose between the inode number of the keys and the number of inodes of the dentry nodes.

Operations

Overall, the complexity of defining operations in Alloy seems similar to the complexity of defining operations in KIV. The additional statements added to facilitate model checking increased the size of the operations. On the one hand, this fact can be taken as a disadvantage since due to the size of the operations. However it can also be taken as an advantage since the user is aware of possible issues in the model. Actually, the main disadvantages are not in the operations themselves but in the operations under data types. Starting by the sequences in Alloy (to represents the journal), the fact that it is only possible to add one element at a time turns out to be a slight disadvantage. In each operation only three addresses are added to the journal which did not cause major problems, using auxiliary variables. In this case, the specification of lists provided KIV take an advantage over the sequences provided by Alloy.

Operations such as *unlink* caused some minor problems when removing some contents of RAM index. The *nodeindex* specification (that represents the RAM index and the flash store in KIV) provides more freedom in their operations: the same operator is used to add a new entry, or to select an entry (or several) identified by a particular key. This can be advantageous but it can also be confusing to understand which operation the operator represents. In Alloy, removing some addresses from the RAM index requires, again, using auxiliary variables. Fortunately, Alloy provides all the necessary operations to perform this procedure: domain restrictions and differences between map relations. Despite being a less elegant solution, it is very clear and it does not create ambiguities.

In KIV, *if-then-else* statements also raised some inconveniences: sometimes it was not easy to determine which instructions were in the boundary of *else*, particularly in the operation *writepage*.

Not needing a top-level operation is another advantage of Alloy. The preconditions are directly

written in the corresponding operation making the process easier and simpler.

Verification

With respect to assertions/theorems used in verification, Alloy takes a great advantage. Despite the translation of theorems to Alloy being literal, the number of assertions needed in the model-checking of the model is much smaller in Alloy than in KIV. As previously mentioned, in Alloy it is not necessary to check for the termination of operations, hence there are about 17 theorems that need not to be defined in Alloy.

There are also other theorems not defined in Alloy because they are not appropriate to the model in Alloy. This is the case of *theory-new1-exists* which ensures that a given address does not exist in the flash store. In Alloy, it is sufficient to ensure this condition as pre-condition.

In KIV, the proofs associated to a certain theorem can be very extensive and complex. For instance, proving *datanode_cons_unlink* involves 90 steps, as shown in the proof tree available in Appendix A.1.15. Despite the proofs being done automatically by KIV, following and interpreting them can be a little confusing, not because of the tool but of the complexity of the model.

In Alloy, assertions are clear and easily readable. The output produced by Alloy Analyzer can be easily readable. For instance, in the metamodel generated to UBIFS, it is trivial to identify not only the map relations and the involved data types but also the relationship between data types.

In the counter-examples, Alloy also identifies the signatures that break the assertion. This way, it is easy to identify which condition violates the invariant. However, in a complex model such as UBIFS, the counter-examples can be very complex and confusing. And in spite of Alloy distinguishing problematic signatures, it can be hard to identify the real issue (even with the help of the Evaluator), mainly when Alloy creates additional atoms.

In general, both tools are very advantageous in spite of having different goals. However, since the verification technique used by KIV is more accurate than the technique used in Alloy, KIV shows advantage over Alloy.

Overview of the model

It is beyond doubt that the abstract specification developed in [36] is very extensive and very detailed. This specification has abstracted from many details of a real flash file system, such as wear leveling and index structures. In fact, KIV provides several concrete data types to build a specification close to the implementation. However, in Alloy most of data types are abstract, i.e., most of data types are signatures and they represent "something". Actually, this is the main idea of Alloy: building a model as abstract as possible. One might think of types such as the integer numbers as being a concrete data type. However, according to Daniel Jackson in [26], adding integer numbers to the language turns out to be not advantageous. Quite often it is enough to postulate one or two basic axioms of such numbers, which capture in fact what is relevant of integer numbers for the particular model in hands. Such concrete types often appear in the problem domain but this does not entail that they should be modeled as such (such is the case of inode numbers in our Alloy model, which

rather than full-fledged integers play the role of unique identifiers). Jackson believes that there is often a more abstract description that is a better representation of the problem. In fact, this is the reason of the limitation of integer numbers in Alloy. Hence, in the Alloy model of UBIFS, the inode numbers contained in the keys do not need be modeled as integers since there is no mathematical operation under them. Therefore, and according to the overall philosophy of Alloy, having such a detailed model is would be an overspecification. The model is too specific and it does not exploit the potential of Alloy (such as point-free notation), a more abstract model would be a better option.

Regarding the assertions and invariants, another option could have been taken: grouping all the invariants into one main invariant could be a better approach, i.e., is it would ensure the consistency of the file system (according to the specified invariants) and would reduce the number of assertions. Grouping all the conditions into one main invariant will bring another advantage. According to the operation *unlink*, when a file or directory is removed, the keys of the corresponding inode node are removed from the RAM index. This condition suggests that the RAM index does not contain keys of deleted nodes. However, there no invariant ensuring this condition, meaning that it is possible that the RAM index stores "deleted" keys. There is not any counter-example associated to this situation, however it might be wrong.

Another doubt about inode nodes with size 0 is that, apparently, they should not be used in operations or invariants (for instance, it can be used by the function *getinode*) but, actually, they can be used by *getinode* meaning that they can be compared, for instance, in the invariant *datanode-cons* or there could be some inode nodes with size 0 that do not represent deleted files or directories. Special attention must be paied to inode nodes with size or number of links 0.

4.1.2 Overview of the tools

Many features have been mentioned of both tools and both have their advantages and disadvantages. There is not any doubt about the strengths of KIV: the specification language is powerful and rich, and it provides enough fredoom to the user to implement several projects of different nature. As said before, in many situations the implementations (specifications and procedures) are intuitive, however sometimes some operators and some axioms can be very confusing. The manual of KIV is of great help. However, it is very long and searching for something turns out to be harder than expected. Currently, KIV is available for Linux. For other operating systems, a virtual machine is needed.

As to the strengths of Alloy, the language is simple, easy to read and write and suited for formal modeling in different domains. Unlike KIV, the documentation is not very organized. In fact, there is no manual documentation of Alloy. There are several small tutorials and FAQ that could help solving small issues and documentation of Daniel Jackson in [26] that helps learning the language.

A very significant advantage of Alloy is that of providing a graphical visualization of the both metamodel and model instances. A graphical view makes the identification of problems (counter-examples or even other issues) easier. Sometimes counter-examples can be very complex and very confusing and it is hard to find where the problem is. In such situations, the Evaluator can be of great use. A detailed model enlarges its graphical visualization. However, this situation occurs

not only because of the complexity of the model but because Alloy Analyzer sometimes generates, by default, unnecessary atoms, that turns out to be a disadvantage.

4.1.3 Difficulties

In this section the main difficulties that arose in this project will be presented. It is divided into three groups: Section 4.1.3 approaches the difficulties that emerged in the tool, the Section 4.1.3 presents all the issues related to the study of the original KIV model and Section 4.1.4 approaches all the issues that emerged during the verification of the model in Alloy.

The KIV tool

Translating KIV to Alloy involved studying in depth the tool KIV: how the tool works, the syntax and the specification language, how proofs are performed and how are they interpreted. Indubitably, KIV is a powerful tool, capable of solving complex problems. However, working with all the features is not alaways intuive if the user is not familiar with the tool. An example of this situation occurs after choosing a specification and choosing the option *Work on...*, the menu *Proof* raised some doubts. Options such as *Continue a partial proof*, *Load a proof* or *Reprove a theorem* were also found a bit dubious. The difficulty lies not in their meaning but in their usage. Knowing how and when should they be used is still unclear. The menu *Simplifier* is also still unclear, probably due to the lack of experience by working with theorem provers. Thus, working with simplifier or elimination rules, or even working with heuristics and theorems is still unknown. In short, the main difficulties lie, in general, in the handling of projects (handling proofs and also creating new projects).

The syntax and the model in KIV

Understanding the syntax and the structure of the abstract specification in KIV also had some issues. Starting with the structure of the abstract specification, the graphic visualization of the model and the information available on [1] do not fulfill the lack of information about the structure of the model. Hence, understanding exactly the structure of the whole project turned out to be a rather arduous task. The first question arose in the definition of *nodes* and *inodes* in KIV. *Nodes* represent the main data structures of the file system, however, the main purpose of the definition *inodes* is still unknown. In Alloy, this data type was taken as an auxiliary data type.

The hierarchy of the specification *addresslist* may be a little complex, since it is an enrichment of several specifications. However, it turned out to be easy to understand the operations over this data type.

Some difficulties also arose in the *filesystem-base* specification. Operations such as *dentrykeys* and *datakeys* are used in some axioms of the current specification. However, knowing exactly how are they performed is unclear without knowing its definition. Both functions appear to be performed respectively under dentry and datakeys. Contrary to expectations, the definition of both functions was not in the specification *key*. After carefully analyzing the involved specifications in *filesytem-base*, the definition of *dentrykeys* and *datakeys* was found in the specification *nodeindex*.

Sometimes, before understanding the relevance of indentation, statements *if-then-else* can be rather confusing. The operation *datanode-cons* is an example of this situation. There existing statement *if-then* is not clear: the lack of parentheses does not delimitate the instruction inside the *then* boundary, causing a wrong definition of the operation in Alloy. Only after considering the behavior of the file system (when a data node is inserted in the file system, an inode node is also inserted) and considering the indentation of the definition in KIV, the definition of the operation was fixed.

As mentioned before, some operators under the *store* specification caused some issues. Since that the same operator may represent two different operations. Hence distinguishing the right operation in the several operations of the file system was not always obvious. Understanding exactly all the changes of the RAM index, for instance, in the operation *unlink* required special attention. It was necessary to identify very clearly what keys should be added to the RAM index and what keys must be removed. The same situation was found in the operation *rename* that caused additional issues: defining such a large operation required special attention, not only due the operators of *store* but also due the large number of existing conditions.

Studying the proofs available online of the file system was also a challenge, even without working directly with tool. Interpreting their meaning was also possible thanks to the invaluable help of Gerhard Schellhorn. Extensive proofs were not studied due their large size.

4.1.4 The verification in Alloy

The main issues lies in the model checking of the file system, especially due the used data structures. Updating the flash store (or the RAM index) with a new node does not entail replacing an old inode by the added inode. Hence, and as mentioned before, the flash store (respectively the RAM index) is updated with new inodes (respectively new keys) that contain existing data. Hence, the RAM index stores different keys with the same inode number and the flash store may contain different inodes with the same key. Duplicated nodes and duplicated keys caused many issues in several assertions, particularly in the assertion *datanode-cons-writepage* as mentioned in Section 3.3.5. The assertions *fs-cons-writepage*, *fs-cons-create* and *fs-cons-mkdir* were also a challenge. Many efforts were spent to solve all problems caused by duplicated nodes (according to Section 3.3.5 this situation occurs after the execution most of operations). The comparison between old and new data (i.e., the comparison between the old nodes and updated nodes) caused several issues, especially when the invariants *fs-file-cons* and *fs-dir-cons* (and consequently *fs-inode-cons*) are ensured. In fact, comparing old nodes to new nodes caused problems in all statements of *fs-file-cons* (its definition is available in Appendix A.2.11): comparing the the size or the number of links to a certain set of keys and comparing the new data nodes inserted to old inode nodes. Hence, writing directly all the necessary conditions for the new inserted data, and for the three problematic operations required much effort and time. It was definitely the most arduous task of modeling the UBIFS in Alloy.

The more complex the model is, the more complex the counter-examples are. When a counter-example has many atoms and relations, identifying the causes of the problem turns out to be hard. This is the case of the UBIFS: most of times, the cause is the large number of atoms and relations

generated by Alloy Analyzer. Sometimes, the used scope is not big enough to generate an instance. That may cause some counter-examples where they do not exist (for instance, integers overflow), increasing the scope easily solves the problem. Hence, special attention was taken to the generated counter-examples and to the scope of the Alloy Analyzer.

Finally, the time taken by the solver to check the assertions also turn out to be an issue and finding the right solver also required some studies about SAT solvers. The best solution found is not, actually, the best option. However, due the impossibility of using the solver *Berkmin*, *minisat* turn out to be the best solution.

4.2 Future Work

Modeling the abstract specification available in [1] was not trivial, it took many efforts and time (several months). Due to its complexity some of the goals previously proposed were not achieved. Beyond the goals not achieved, there are several approaches that can be taken under the UBIFS: submitting the abstract specification to another tool (and formal method technique), building a simpler and more abstract model of the file system using Alloy or even building a model and using the ESC-PF technique in the verification of the model. The following subsections describe, in more detail, each approach mentioned above.

4.2.1 VDM model of UBIFS

Following the work of Miguel Ferreira in [17], it might be interesting to submit the UBIFS file system to modeling in VDM . Since VDM++ includes support for testing and proving properties of models and generating program code from validated VDM models, the idea of comparing the work developed in KIV and the work that can be done using VDM might have many advantages: on one hand, comparing KIV tools to VDM tools might be interesting. Besides emphasizing the KIV tool and the developed abstract specification, a different experience with KIV could produce different conclusions.

On the other hand, the idea of submitting the abstract specification to a different verification technique could also have interesting results:

- What problems could arise either in the specification or in the verification?
- Once VDM++ includes testing support, what kinds of tests could be done?
- The advantages/disadvantages of both models and techniques;
- The advantages/disadvantages of the program code generated of VDM++.

Modeling the abstract specification in VDM-SL/VDM++ could also have some advantages related to the data types provided by VDM. Like KIV, VDM also provides basic data types: booleans, natural numbers, integer numbers, characters, etc. In this way, the problem of the integers presented in Alloy might be solved in VDM. Apparently, basic data types could also make the translation between KIV and VDM easier due to the similarities of the languages.

VDM++ also provides collection types such as mappings, sets as sequences. These data types have a similar behavior to the corresponding data types in Alloy: the structure is similar and similar operators (such as unions, overwriting or the cardinality of a set) are also available. However, VDM tool does not provide a graphical overview of the model and it also does not provide a graphical visualization of potential issues that may appear. Thus, doing a comparative work between both models may answer the following questions:

- What were the main differences/similarities between both specifications?
- Could it be done a different specification of the UBIFS using VDM?

The challenge is launched. It might be a small challenge or it could turning out to be a huge challenge. Regardless of the size of the challenge, it also is a contribution to the mini challenge proposed in [29].

4.2.2 A more abstract model of UBIFS using Alloy

According to research done, this is the first model developed in Alloy of the UBIFS, and according to Section 4.1, the developed model is very detailed and complex. Thus, building a different version of the UBIFS may be an appealing challenge. The main idea is to use Alloy to do a comparative work between the current version of the model and a new built model. The main goal is trying to build a more abstract and simples model of the UBIFS solving the following issues:

- building a simpler version of the UBIFS entail loosing precision, i.e. building a model too unreal?
- what advantages/disadvantages could arise from an abstract version of the file system?
- could be the new version of UBIFS specified using other tools/techniques? What are the expected results?

In the current thesis, the main concern is modeling the UBIFS file system for flash memories. Currently, there are several models in Alloy of several file systems. Particularly, in [30], Daniel Jackson and Eunsuk Kang designed a flash file system using Alloy. The main idea of their work is emphasizing key concepts of Alloy, using as an example, the construction and analysis of a design for a flash file system. Their model includes not only the basic operations, but also crucial features to NAND flash memory that contribute to increased complexity of the file system (wear leveling technique and erase-unit reclamation). Their design also addresses the issues of fault-tolerance by providing a mechanism for recovering from unexpected hardware failures.

Based on the work of Daniel Jackson and Eunsuk Kang, it may be interesting to compare a new design of the UBIFS to their work. The main goal is exploring the several features of Alloy and several designing techniques. Finally, the results of the design analysis of both models could also be discussed: the differences, the similarities and the features of the models.

This work could also be compared to the abstract specification of KIV. Basically, it would be interesting to see how a design (in this case, of a file system) can be influenced by the used tool, i.e.:

- Does writing an abstract specification in KIV entail building a detailed model?
- Can a abstract model in Alloy be reproduced in KIV, keeping the simplicity?

Building a more abstract model also could raise additional advantages, such as having the benefits of using the *point-free* notation. Subsection 4.2.3 will present further details about modeling a file system using PF notation.

4.2.3 Using ESC-PF notation in the UBIFS

As mentioned before, designing a more abstract file system will take advantage of *point-free* notation. Due the possibility of using PF notation in Alloy, the main goal is building an abstract model to reduce tool integration costs. The work in Miguel Ferreira and José N. Oliveira of [16] focus on the integration of different formal methods and tools in a tool-chain for modeling and verification, including Alloy and relational algebra. Their work shows why Alloy is a suited candidate for integration with relational algebra (using PF notation) and how the translation can be done. Since the developed work also focus in the verification of a journaled file system, it might be interesting to submit the UBIFS file system to the same verification technique. A comparative work could be done between the journaled file system of [16] and the UBIFS file system, that involves:

- showing the differences and similarities of both models: data types, operations, etc;
- describing what difficulties might arise using ESC-PF;
- enumerating the advantages/disadvantages of applying the tool-chain to the UBIFS file system .

According to Miguel Ferreira and José N. Oliveira, matching model-checking in Alloy with manual proofs carried out in the PF algebra of binary relations emphasizes the positive impact of the lemma “everything is a relation” on software verification. Hence, using Alloy model-checking and ESC-PF to formally verify the UBIFS appears to be an attractive challenge and it also would be a contribution to the mini challenge proposed in [29].

Bibliography

[1] Abstract Specification of the UBIFS File System for Flash Memory home page. `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`.

[2] The open group, the posix 1003.1, 2003 edition specification, available online. `http://www.opengroup.org/certification/idx/posix.html`.

[3] June Andronick, Boutheina Chetali, and Christine Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2005.

[4] Stefano Bacherini, Alessandro Fantechi, Matteo Tempestini, and Niccolò Zingoni. A story about formal methods adoption by a railway signaling manufacturer. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2006.

[5] Christel Baier and Joost P. Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[7] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *Operating Systems Review*, 41(2):88–93, 2007.

[8] Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In Patrice Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2005.

[9] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[10] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.

[11] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[12] E.M. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, page 54. Springer, 1997.

[13] Alcino Cunha and Jorge Sousa Pinto. Making the point-free calculus less pointless. 2004.

[14] R. Drechsler and S. Höreth. Gatecomp: Equivalence checking of digital circuits in an industrial environment. In *Int'l Workshop on Boolean Problems*, pages 195–200. Citeseer, 2002.

[15] Rainer Drexler, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, Werner Stephan, and Andreas Wolpers. The kiv system: A tool for formal program development. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *STACS*, volume 665 of *Lecture Notes in Computer Science*, pages 704–705. Springer, 1993.

[16] M.A. Ferreira and J.N. Oliveira. Variations on an Alloy-centric tool-chain in verifying a journaled file system model, 2010. (Submitted to Springer's Formal Aspects of Computing).

[17] Miguel Alexandre Ferreira. Verifying Intel® Flash File System Core. Master's thesis, Minho University, Jan. 2009.

[18] Miguel Alexandre Ferreira and José Nuno Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *SBMF*, volume 5902 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2009.

[19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[20] T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI-Unsorted Block Images.

[21] Holger Grandy, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, and Wolfgang Reif. Verification of mondex electronic purses with kiv: From a security protocol to verified code. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2008.

[22] Anthony Hall. Correctness by construction: Integrating formality into a commercial development process. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2002.

[23] Dominik Haneberg, Gerhard Schellhorn, Holger Grandy, and Wolfgang Reif. Verification of mondex electronic purses with kiv: from transactions to a security protocol. *Formal Asp. Comput.*, 20(1):41–59, 2008.

[24] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 78–78. Springer, 2005.

[25] A. Hunter. A brief introduction to the design of ubifs. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.

[26] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Heyward Street, Cambridge, MA02142, USA, April 2006.

[27] Jawahar Jain, Amit Narayan, Masahiro Fujita, and Alberto L. Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *ICCD*, pages 445–454, 1997.

[28] Neil D. Jones and Markus Müller-Olm, editors. *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*. Springer, 2009.

[29] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Asp. Comput.*, 19(2):269–272, 2007.

[30] Eunsuk Kang and Daniel Jackson. Designing and analyzing a flash file system with alloy. *Int. J. Software and Informatics*, 3(2-3):129–148, 2009.

[31] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[32] Catherine Meadows. Formal verification of cryptographic protocols: A survey. In Josef Pieprzyk and Reihaneh Safavi-Naini, editors, *ASIACRYPT*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1994.

[33] José N. Oliveira. Transforming data by calculation. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 134–195. Springer, 2007.

[34] José N. Oliveira. Extended static checking by calculation using the pointfree transform. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *LerNet ALFA Summer School*, volume 5520 of *Lecture Notes in Computer Science*, pages 195–251. Springer, 2008.

[35] G.K. Palshikar. An introduction to model checking. *Embedded Systems Programming*, 17(3):22–29, 2004.

[36] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the ubifs file system for flash memory. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 190–206. Springer, 2009.

[37] Ann E. Kelley Sobel and Michael R. Clarkson. Response to "comments on 'formal methods application: An empirical tale of software development'". *IEEE Trans. Software Eng.*, 29(6):572–575, 2003.

[38] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *FAST*. USENIX, 2003.

[39] Geoff Sutcliffe and Christian B. Suttner. Evaluating general purpose automated theorem proving systems. *Artif. Intell.*, 131(1-2):39–54, 2001.

Appendix A

Models

A.1 KIV Model

A.1.1 Node specification

node =
data specification
  using nat, string-data, page, key
  node :=
    inodenode (. .key : key ; . .directory : bool ; . .nlink : nat ; . .size : nat ;) **with** . .inode?
    dentrynode (. .key : key ; . .name : string ; . .ino : nat ;) **with** . .dentry?
    datanode (. .key : key ; . .data : page ;) **with** . .data?
  ;
  variables nd, nd0, nd1, nd2, nd3, node, node0, node1, node2, node3 : node ;
end data specification

A.1.2 Key specification

key =
data specification
  using nat, string-data
  key :=
    inodekey (. .ino : nat ;) **with** . .inode?
    datakey (. .ino : nat ; . .part : nat ;) **with** . .data?
    dentrykey (. .ino : nat ; . .name : string ;) **with** . .dentry?
  ;
  variables key: key;
end data specification

A.1.3 Dentry specification

dentry =
data specification
  using nat, string-data
  dentry :=
    mkdentry (. .name : string ; . .ino : nat ;) **with** . .dentry?
    negdentry (. .name : string ;) **with** . .negdentry?
  ;
  variables dentry: dentry;
end data specification

A.1.4 Store specification

store =
generic specification
  parameter elemdata **using** nat **target**
  sorts store, elemdata;
  ...
end generic specification

A.1.5 Flash store specification

nodestore =
actualize store **with** node, address **by morphism**

store $\rightarrow$ nodestore; data $\rightarrow$ node; elem $\rightarrow$ address; a $\rightarrow$ adr; b $\rightarrow$ adr0;
c $\rightarrow$ adr1; a0 $\rightarrow$ adr2; d $\rightarrow$ nd; d0 $\rightarrow$ nd0; d1 $\rightarrow$ nd1; d2 $\rightarrow$ nd2

end actualize

A.1.6 RAM and flash index specification

nodeindexa =
actualize store **with** key, address **by morphism**

store $\rightarrow$ nodeindex; elem $\rightarrow$ key; data $\rightarrow$ address; elemdata $\rightarrow$ keyaddressdata;
a $\rightarrow$ key; b $\rightarrow$ key0; c $\rightarrow$ key1; a0 $\rightarrow$ key2; d $\rightarrow$ adr; d0 $\rightarrow$ adr0; d1 $\rightarrow$ adr1; d2
$\rightarrow$ adr2; st $\rightarrow$ ni; st0 $\rightarrow$ ni0; st1 $\rightarrow$ ni1; st2 $\rightarrow$ ni2; elemdatavar $\rightarrow$ keyaddressvar

end actualize

A.1.7 List-dup specification

list-dup =
enrich list **with**
  functions

. ++ .	:	list × elem	→	list	**prio** 9 **left**;
rmdup	:	list	→	list	;

  predicates

dups	:	list;
disj	:	list × list;

A.1.8 Addresslist specification

addresslist =
actualize list-dup **with** address **by morphism**

list → addresslist; elem → address; a → adr; b → adr0; c → adr1; a0 → adr2;
x → ax; y → ay; z → az; x0 → ax0; y0 → ay0; z0 → az0; x1 → ax1; y1 → ay1;
z1 → az1; x2 → ax2; y2 → ay2; z2 → az2

end actualize

A.1.9 Filesystem-base specification

filesystem-base =
enrich inode, dentry, nodestore, addresslist, file, nodeindex **with**
  functions

getinode	:	nat × nodestore × nodeindex	→	inode	;
links	:	nat × nodestore × nodeindex	→	keylist	;
subdirs	:	nat × nodestore × nodeindex	→	keylist	;

  predicates

log-cons	:	nodestore × nodeindex × nodeindex × addresslist;
fs-cons	:	nodestore × nodeindex;
store-cons	:	nodestore × nodeindex × nodeindex × addresslist;
datanode-cons	:	nodestore × nodeindex;
nodekey-cons	:	nodestore × nodeindex;
key-iso	:	key × nodestore × nodeindex × nodestore × nodeindex;
fs-key-cons	:	key × nodestore × nodeindex;
fs-dir-cons	:	key × nodestore × nodeindex;
fs-inode-cons	:	key × nodestore × nodeindex;
fs-file-cons	:	key × nodestore × nodeindex;
fs-data-cons	:	key × nodestore × nodeindex;
fs-link-cons	:	key × nodestore × nodeindex;
fs-dentry-cons	:	key × nodestore × nodeindex;
valid-ino	:	nat × nodestore × nodeindex;
valid-dir-ino	:	nat × nodestore × nodeindex;
valid-file-ino	:	nat × nodestore × nodeindex;
valid-dentry	:	dentry × nodestore × nodeindex;
valid-dentry	:	nat × dentry × nodestore × nodeindex;
valid-negdentry	:	nat × dentry × nodestore × nodeindex;
valid-file	:	file × nodestore × nodeindex;
valid-dir	:	file × nodestore × nodeindex;

variables

fs, fs1, fs2, st: nodestore;
ri, ri1, ri2, fi, fi1, fi2: nodeindex;
log: addresslist;
adr, ad, ad0, ad1, ad2, ad3: address;
dent: dentry;
key, key2: key;
ks: keylist;
file: file;
ino: nat;

axioms

store-cons : ⊢ store-cons(fs, ri, fi, log) ↔ (∀ adr. adr ∈ log ∨ adr ∈ ri → adr ∈ fs);

datanode-cons : ⊢ datanode-cons(fs, ri) ↔ ∀ key. key.data? ∧ key ∈ ri ∀ key. → valid-file-ino(key.ino, fs, ri) ∧ key.part < getinode(key.ino, fs, ri).size;

nodekey-cons : ⊢ nodekey-cons(fs, ri) ↔ (∀ key. key ∈ ri → ri[key] ∈ fs ∧ fs[ri[key]].key = key);

links : ⊢ key ∈ links(ino, fs, ri) ↔ key ∈ ri ∧ key.dentry? ∧ fs[ri[key]].ino = ino;

links-nodup : ⊢ ¬ dups(links(ino, fs, ri));

subdirs-nodup : $\vdash \neg$ dups(subdirs(ino, fs, ri));

getinode : $\vdash$ getinode(ino, fs, ri) = mkinode(ino, fs[ri[inodekey(ino)]].directory, fs[ri[inodekey(ino)]].nlink, fs[ri[inodekey(ino)]].size);

fs-cons : $\vdash$ fs-cons(fs, ri) $\leftrightarrow$ ($\forall$ key. key $\in$ ri $\rightarrow$ fs-key-cons(key, fs, ri));

fs-key-cons :

$\vdash$ fs-key-cons(key, fs, ri) $\leftrightarrow$ ri[key] $\in$ fs $\wedge$ fs[ri[key]].key = key
$\wedge$ (key.dentry? $\rightarrow$ fs-dentry-cons(key, fs, ri))
$\wedge$ (key.data? $\rightarrow$ fs-data-cons(key, fs, ri))
$\wedge$ (key.inode? $\rightarrow$ fs-inode-cons(key, fs, ri));

valid-ino : $\vdash$ valid-ino(ino, fs, ri) $\leftrightarrow$ inodekey(ino) $\in$ ri $\wedge$ ri[inodekey(ino)] $\in$ fs $\wedge$ ino $\neq 0$;

valid-file-ino : $\vdash$ valid-file-ino(ino, fs, ri) $\leftrightarrow$ inodekey(ino) $\in$ ri $\wedge$ ri[inodekey(ino)] $\in$ fs $\wedge$ $\neg$ getinode (ino, fs, ri).directory $\wedge$ getinode (ino, fs, ri).nlink > 0 $\wedge$ ino $\neq 0$;

inodenode-ino : $\vdash$ inodenode(key, boolvar, n0, n1).ino = key.ino;

end enrich

A.1.10 Filesystem-asm specification

FSASM# =
asm specification filesystem-asm
using filesystem-base
procedures
FSASM# : nodestore × nodeindex × nodeindex × addresslist;
FSOP# : nodestore × nodeindex × nodeindex × addresslist;
lookup# nat × nodestore × nodeindex : dentry;
create# nat : dentry × nodestore × nodeindex × addresslist;
unlink nat : dentry × nodestore × nodeindex × addresslist;
...

variables
FS, FS2, fs, fs2: nodestore;
RI, FI, RI2, FI2, ri, fi, ri2, fi2: nodeindex;
LOG, LOG2, log, log2: addresslist;
OLD_INO, P_INO, NEW_INO, PAGENO, INO, old_ino, p_ino, new_ino, pageno, ino: nat;
P_INODE, INODE, OLD_INODE, NEW_INODE, p_inode, inode: inode;
DENT, OLD_DENT, P_DENT, NEW_DENT, dent, old_dent, p_dent, new_dent: dentry;
PAGE, page: page;

```
        FILE, file: file;
        ND, nd: node;
        ADR, ADR1, ADR2, ADR3, ADR4, ADR5, ADDRESS, adr, adr1, adr2, adr3, adr4,
        address: address;
        KS, ks: keylist;
        is_dir, old_dir, new_dir: bool;
    state variables FS, RI, FI, LOG;
    initial state FS = ∅ ∧ RI = ∅ ∧ FI = ∅ ∧ LOG = []
    final state false
    asm rule FSOP#
    declaration

FSASM#(FS, RI, FI, LOG) { while ¬ false do FSOP# };

FSOP# (FS, RI, FI, LOG) {
    choose P_INO, DENT with valid-dentry(P_INO, DENT, FS, RI) ∧ valid-file-ino(DENT.ino,
            FS, RI) in unlink ifnone skip
    ∨ choose P_INO, DENT with valid-dir-ino(P_INO, FS, RI) ∧ valid-negdentry(P_INO,
            DENT, FS, RI) in mkdir# ifnone skip
    ...
    ∨ choose FILE, PAGENO, PAGE with valid-file(FILE, FS, RI) in writepage ifnone skip
    ...
};

unlink(P_INO; DENT, FS, RI, LOG) {
    let P_INODE = getinode(P_INO, FS, RI), INODE = getinode(DENT.ino, FS, RI) in
      let node = inodenode(inodekey(INODE.ino), INODE.directory, INODE.nlink - 1,
        INODE.size)
        in choose ADR1, ADR2, ADR3 with new(ADR1, ADR2, ADR3, FS) in {
          in FS := FS[ADR1, dentrynode(dentrykey(P_INO, DENT.name), DENT.name, 0)]
                  [ADR2, inodenode(inodekey(P_INODE.ino), P_INODE.directory,
                        P_INODE.nlink, P_INODE.size - 1)]
                  [ADR3, node],
          LOG := LOG + ADR1 + ADR2 + ADR3;
          RI := RI – dentrykey(P_INO, DENT.name);
          RI[inodekey(P_INO)] := ADR2;
          if INODE.nlink = 1
            then { RI := RI – inodekey(INODE.ino); do_delete#}
            else RI[inodekey(INODE.ino)] := ADR3
};
    DENT := negdentry(DENT.name)
};
```

```
writepage(FILE, PAGENO, PAGE; FS, RI, LOG) {
        if getinode(FILE.ino, FS, RI).size ⩽ PAGENO then
        choose ADDRESS with new(ADDRESS, FS)
        in let INODE = getinode(FILE.ino, FS, RI)
          in{
            FS := FS[inodenode(inodekey(FILE.ino), INODE.directory, INODE.nlink
            , PAGENO + 1), ADDRESS]
            LOG := LOG + ADDRESS;
            RI := RI[inodekey(FILE.ino), ADDRESS]
          };
        choose ADDRESS with new(ADDRESS, FS)
        in {
           FS := FS[datanode(datakey(FILE.ino, PAGENO), PAGE), ADDRESS]
           LOG := LOG + ADDRESS;
           RI := RI[datakey(FILE.ino, PAGENO), ADDRESS]
        }
};
```

end asm specification

A.1.11 Replay process

```
replay#(FS, FI, LOG; var RI) {
      RI := FI;
      let LOG2 = LOG in
        while LOG2 ≠ [] do {
             let adr = LOG2.first in
                replayone#;
                LOG2 := LOG2.rest
        }
};
```

A.1.12 File system Specification (log-cons invariant)

```
filesystem=
        enrich filesystem-asm with
          axioms

     log-cons : log-cons(fs, ri, fi, log) ⇔ ⟨|replay# |⟩iso(fs, ri, fs, ri2);

end enrich
```

A.1.13 Log-cons invariant of the unlink operation

logcons-unlink:

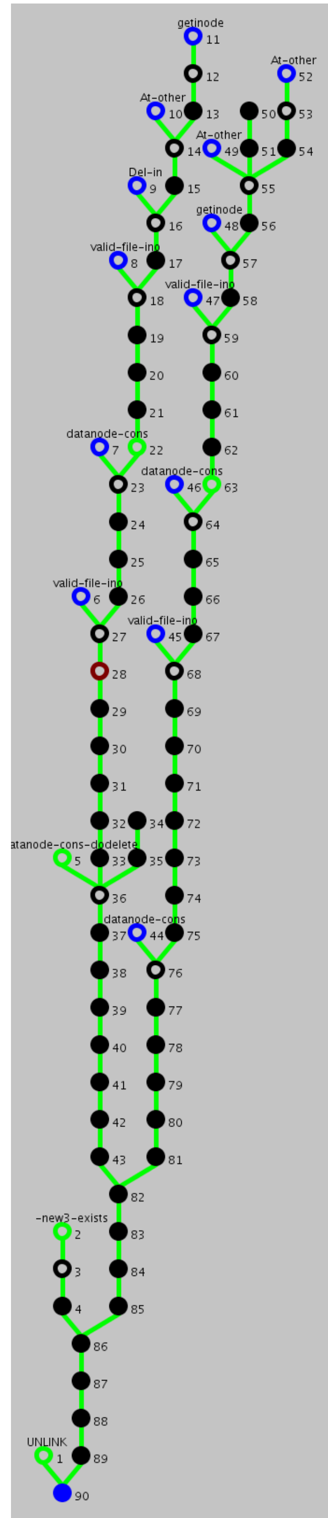
log-cons(fs, ri, fi, log),
valid-dentry(P_INO, DENT, fs, ri),
valid-file-ino(DENT.ino, fs, ri),
store-cons(fs, ri, fi, log),
datanode-cons(fs, ri) $\vdash$ ⟨|unlink|⟩log-cons(fs, ri, fi, log)

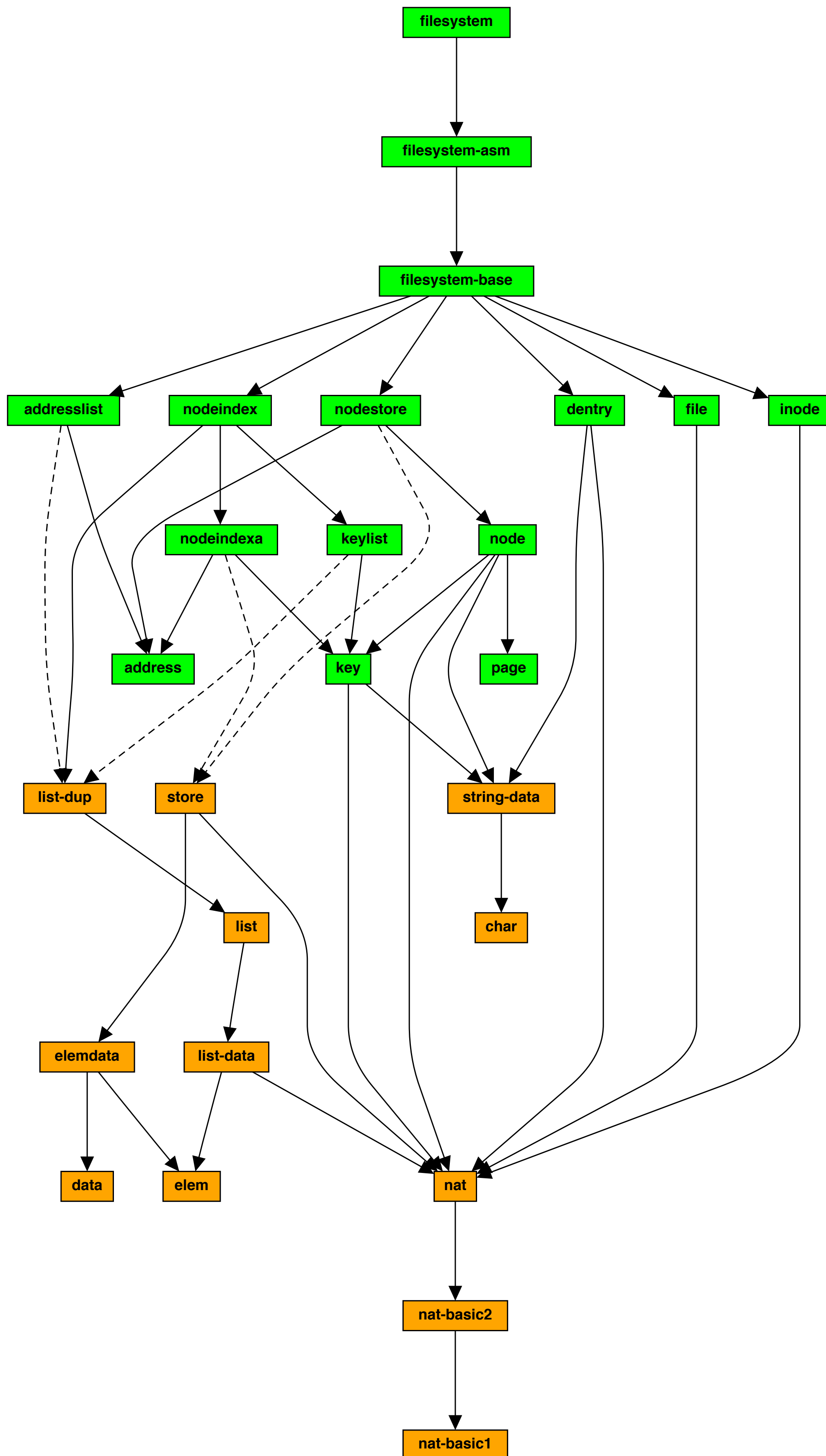
A.1.14 Datanode-cons invariant of the writepage operation

datanode-cons(FS, RI),
valid-file-ino(FILE.ino, FS, RI) $\vdash$ ⟨|writepage|⟩datanode-cons(FS, RI)

A.1.15 Proof tree of datanode_cons_unlink

A.1.16 Graphical overview of UBIFS


getinode
At-other
At-other
At-other
Del-in
getinode
valid-file-ino
valid-file-ino
datanode-cons
datanode-cons
valid-file-ino
valid-file-ino
atanode-cons-dodelete
datanode-cons
-new3-exists
UNLINK


filesystem
filesystem-asm
filesystem-base
addresslist
nodeindex
nodestore
dentry
file
inode
nodeindexa
keylist
node
address
key
page
list-dup
store
string-data
list
char
elemdata
list-data
data
elem
nat
nat-basic2
nat-basic1

A.2 Alloy Model

A.2.1 Node signature

```
sig Node{}

sig Inodenode extends Node {
    ino : Int,
    directory : one Bool,
    nlink : Int,
    size : Int
}{ key in Inodekey}

sig Dentrynode extends Node {
    name : Name,
    dino : Int
}{ key in Dentrykey }

sig Datanode extends Node {
    data : Page
}{ key in Datakey }
```

A.2.2 Key signature

```
abstract sig Key {
     ino: Int
}

sig Inodekey extends Key {}

sig Datakey extends Key {
     part_: Int
}

sig Dentrykey extends Key {
    name: Name
}
```

A.2.3 UBIFS signature

```
sig UBIFS {
     fs : Address →lone Node,
     ri : Key →lone Address,
     log : seq Address,
```

```
    fi : Key →lone Address
}
```

A.2.4 Dentry signature

```
abstract sig Dentry {
  name : one Name
}

sig Mkentry extends Dentry {
    ino : Int
}

sig Negdentry extends Dentry {
}
```

A.2.5 Function getinode

```
fun getinode [ino_ : Int, fs : Address →lone Node, ri : Key →lone Address] : lone Node
{
    fs[Key·({ k : Key | k in ri·dom and k·ino in ino_ } <:ri)]
}
```

A.2.6 Constructors in Alloy

```
fun inodenode_c [ink : Inodekey, dir : Bool, nlink_ : Int, size_ : Int] : one Inodenode
{
  { inn : Inodenode | inn·key in ink and inn·directory in dir and inn·nlink in nlink_ and inn·size in size_ }
}

fun mkinode_c [ino_ : Int, dir : Bool, nlink_ : Int, size_ : Int] : one Inodenode
{
  { inode : Inodenode | inode·key·ino in ino_ and inode·directory in dir and inode·nlink in nlink_ and inode·
      size in size_ }
}
```

A.2.7 Predicate new_inodekey

```
pred new_inodekey [ink : Inodekey, ri : Key →lone Address, fs : Address →lone Node]
{
  ink·ino > 0 and ink·ino not in ri·dom·ino and
  ink·ino not in fs·rng·key·ino and ink·ino not in fs·rng·dino
```

```
}
```

A.2.8 Predicate valid-ino

```
pred valid_ino [ino_ : Int, fs : Address →lone Node, ri : Key →lone Address]
{
  some ink : Inodekey | ink·ino in ino_ and ink in ri·dom ⇒ ri[ink] in fs·dom and fs[ri[ink]]·key in ink
  ino_ not in 0
}
```

A.2.9 Predicate fs_key_cons

```
pred fs_key_cons [key_ : Key, fs : Address →lone Node, ri : Key →lone Address]
{
  ri[key_] in fs·dom and fs[ri[key_]]·key in key_
  key_ in Dentrykey ⇒ fs_dentry_cons[key_, fs, ri]
  key_ in Datakey ⇒ fs_data_cons[key_, fs, ri]
  key_ in Inodekey ⇒ fs_inode_cons[key_, fs, ri]
}
```

A.2.10 Predicate fs-dentry-cons

```
pred fs_dentry_cons[key_ : Dentrykey, fs : Address →lone Node, ri : Key →lone Address]
{
  some n : Inodenode | n in getinodes[key_·ino, fs, ri] and valid_ino [n·key·ino, fs, ri] and n·directory in
     True and n·nlink > 1
  key_·name in fs[ri[key_]]·name
  valid_ino[fs[ri[key_]]·dino, fs, ri]
}
```

A.2.11 Predicate fs-file-cons

```
pred fs_file_cons [key_ : Inodekey, fs : Address →lone Node, ri : Key →lone Address]
{
  some inn : Inodenode | inn·key in key_ and inn in fs·rng and inn·nlink in (#links[key_·ino, fs, ri])
  all key2 : Datakey | key2 in datakeys[ri, key_·ino] ⇒ key2·part_ < fs[ri[key_]]·size
}
```

A.2.12 Predicate fs-dir-cons

```
pred fs_dir_cons_aux [key_ : Inodekey, fs : Address →lone Node, ri : Key →lone Address, fs' : Address →
    lone Node, ri' : Key →lone Address]
{
  some inn : Inodenode | inn·key in key_ and inn in fs·rng and key_ in ri·dom ⇒
  {
    inn·nlink in (#subdirs[key_·ino, fs', ri'] + 2)
    inn·size in #dentrykeys [ri', key_·ino]
  }
  #links[key_·ino, fs', ri'] < 2

}
```

A.2.13 Predicate datanode-cons

```
pred datanode_cons [fs : Address →lone Node, ri : Key →lone Address]
{
  all k : Datakey | k in ri·dom and k in fs·rng·key ⇒ valid_file_ino[k·ino, fs, ri] and
                              k·part_ < getinode[k·ino, fs, ri]·size
}
```

A.2.14 Predicate nodekey-cons

```
pred nodekey_cons [fs : Address →lone Node, ri : Key →lone Address]
{
  all key_ : Key | key_ in ri·dom ⇒ ri[key_] in fs·dom and fs[ri[key_]]·key in key_
}
```

A.2.15 Operation unlink

```
pred UNLINK [p_ino : Int, dent : Dentry, dent' : Negdentry, u, u' : UBIFS]
{
    // pre-conditions
    valid_dentry_with_parent [p_ino, dent, u·fs, u·ri]
    valid_file_ino [dent·ino, u·fs, u·ri]
    // post-conditions
    let p_inode = getinode[p_ino, u·fs, u·ri],
        inode = getinode[dent·ino, u·fs, u·ri] |
      {
        one disj adr1, adr2, adr3 : Address |
          {
            dent' = negdentry_c[dent·name]
```

```
            (adr1 + adr2 + adr3) not in u·fs·dom

            let dn1 = dentrynode_c[dentrykey_c[p_ino, dent'·name], dent'·name, 0],
                in1 = inodenode_c[inodekey_c[p_ino], p_inode·directory, p_inode·nlink, inode·size−1],
                node = inodenode_c [inodekey_c[dent·ino], inode·directory, inode·nlink−1, inode·size],
                dnks = dentrykeys_name [u·ri, p_ino, dent·name] |
              {
                dn1 not in rng[u·fs]
                in1 not in rng[u·fs]
                node not in rng[u·fs]

                u'·fs = u·fs + (adr1 →dn1)
                        + (adr2 →in1)
                        + (adr3 →node)
                u'·fi = u·fi
                let temp = (dnks <:u·ri ),
                    ri_temp = u·ri − temp + (in1·key →adr2) |
                     node·nlink = 1⇒ {let aux = (node·key <:u·ri) |
                                     u'·ri = ri_temp − aux and DO_DELETE[p_inode, u, u']}
                                  else {u'·ri = ri_temp + (node·key →adr3)}
                let s1 = sq/add[u·log, adr1], s2 = sq/add[s1, adr2] | u'·log = sq/add[s2, adr3]
              }
          }
      }
}
```

A.2.16 Operation writepage (old version)

```
pred WRITEPAGE [file : File, pageno : Int, page : Page, u, u' : UBIFS]
{
  // pre-conditions
  valid_file[file, u·fs, u·ri]
  // post-conditions
  let inode = getinode [file·ino, u·fs, u·ri] |
  {
    one disj adr1, adr2 : Address |
    {
    (adr1 + adr2) not in dom[u·fs]

    let dk = datakey_c[file·ino, pageno],
        node = inodenode_c[inodekey_c[file·ino], inode·directory, inode·nlink, (pageno+ 1)],
        dn = datanode_c[dk, page] |
      {
        node not in rng[u·fs]
        dn not in rng[u·fs]
```

```
        (inode·size < = pageno) ⇒
        {
        u'·fs = u·fs + (adr1 →node) + (adr2 →dn)
        u'·ri = u·ri + (node·key →adr1) + (dn·key →adr2)
        u'·fi = u·fi
        let log_temp = sq/add[u·log, adr1] | u'·log = sq/add[log_temp, adr2]
        }
        else
        {
        u'·fs = u·fs + (adr2 →dn)
        u'·ri = u·ri + (dn·key →adr2)
        u'·fi = u·fi
        u'·log = sq/add[u·log, adr2]
        }
      }
    }
  }
}
```

A.2.17 Operation writepage (fixed version)

```
pred WRITEPAGE [file : File, pageno : Int, page : Page, u, u' : UBIFS]
{
  // pre-conditions
  valid_file[file, u·fs, u·ri]

  // post-conditions
  let inode = getinode [file·ino, u·fs, u·ri] |
  {
    one disj adr1, adr2 : Address |
    {
    (adr1 + adr2) not in dom[u·fs]

    let dk = datakey_c[file·ino, pageno],
       node = inodenode_c[inodekey_c[file·ino], inode·directory, inode·nlink, (pageno+ 1)],
       dn = datanode_c[dk, page] |
       {
        node not in rng[u·fs]
        dn not in rng[u·fs]
        (inode·size < = pageno)

        u'·fs = u·fs + (adr1 →node) + (adr2 →dn)
        u'·ri = u·ri + (node·key →adr1) + (dn·key →adr2)
        u'·fi = u·fi
```

```
      let log_temp = sq/add[u·log, adr1] | u'·log = sq/add[log_temp, adr2]
      }
    }
  }
}
```

A.2.18 Operation create

```
pred CREATE [p_ino : Int, dent : Negdentry, dent' : Mkentry, u, u' : UBIFS]
{
  // pre-conditions
  valid_dir_ino [p_ino, u·fs, u·ri]
  valid_negdentry[p_ino, dent, u·fs, u·ri]

  // post-conditions
  let inode = getinode [p_ino, u·fs, u·ri] |
  {
    one disj adr1, adr2, adr3 : Address |
    {

      one ink : Inodekey |
      {
      (adr1 + adr2 + adr3) not in dom[u·fs]

      new_inodekey[ink, u·ri, u·fs]

      dent' = mkdentry_c [dent·name, ink·ino]

      let inode1 = inodenode_c[inodekey_c[p_ino], inode·directory, inode·nlink, (inode·size+ 1)],
          inode2 = inodenode_c[ink, False, 1, 0],
          dn = dentrynode_c[dentrykey_c[p_ino, dent'·name], dent'·name, dent'·ino] |
      {
        inode1 not in u·fs·rng
        inode2 not in u·fs·rng
        dn not in u·fs·rng
        inode1·nlink > 1

        u'·fs = u·fs + (adr1 →inode2) + (adr3 →inode1) + (adr2 →dn)
        u'·ri = u·ri + (inode2·key →adr1) + (inode1·key →adr3) + (dn·key →adr2)
        u'·fi = u·fi
        let s1 = sq/add[u·log, adr1], s2 = sq/add[s1, adr2] | u'·log = sq/add[s2, adr3]
      }
    }
  }
 }
```

```
}
```

A.2.19 Assertion nodekey_cons_unlink

```
pred nodekey_cons [fs : Address →lone Node, ri : Key →lone Address]
{
  all key_ : Key | key_ in ri·dom ⇒ ri[key_] in fs·dom and fs[ri[key_]]·key in key_
}
```

A.2.20 Assertion danode_cons_writepage

```
assert datanode_cons_writepage{
  all u, u' : UBIFS, file : File, pageno : Int, page : Page |
      store_cons [u·fs, u·ri, u·log] and
      valid_file_ino [file·ino, u·fs, u·ri] and
      datanode_cons [u·fs, u·ri] and
      WRITEPAGE [file, pageno, page, u, u'] ⇒ {lone inn : Inodenode | inn in u'·fs·rng and inn not in u·fs·
    rng and {some k : Datakey | k in u'·ri·dom and k in u'·fs·rng·key ⇒ valid_file_ino [k·ino, u'·fs, u'·ri] and
     k·part_ < inn·size} }
}
```

A.2.21 Assertion prepost_unlink

```
assert prepost_unlink{
  all u, u' : UBIFS, p_ino : Int, dent, dent' : Dentry |
      valid_dentry_with_parent [p_ino, dent, u·fs, u·ri] and
      valid_file_ino [dent·ino, u·fs, u·ri] and
      UNLINK [p_ino, dent, dent', u, u'] ⇒ valid_negdentry [p_ino, dent', u'·fs, u'·ri]
}
```

A.2.22 Assertion prepost_readpage

```
assert prepost_readpage {
  all u, u' : UBIFS, pageno : Int, page : Page, file : File |
  valid_file [file, u·fs, u·ri] and
  READPAGE [file, pageno, page, u, u'] ⇒ {(page not in Emptypage) ⇒ {some dn : Datanode | dn in rng[u
    ·fs] and dn·key·part_ in pageno ⇒ dn·data in page}}
}
```

A.2.23 Assertion prepost_writepage

```
assert prepost_writepage {
  all u, u' : UBIFS, pageno : Int, page : Page, file : File |
  valid_file [file, u·fs, u·ri] and
  WRITEPAGE [file, pageno, page, u, u'] ⇒ {some dn : Datanode | dn in rng[u'·fs] and dn·data in page
     and dn·key·part_ in pageno}
}
```

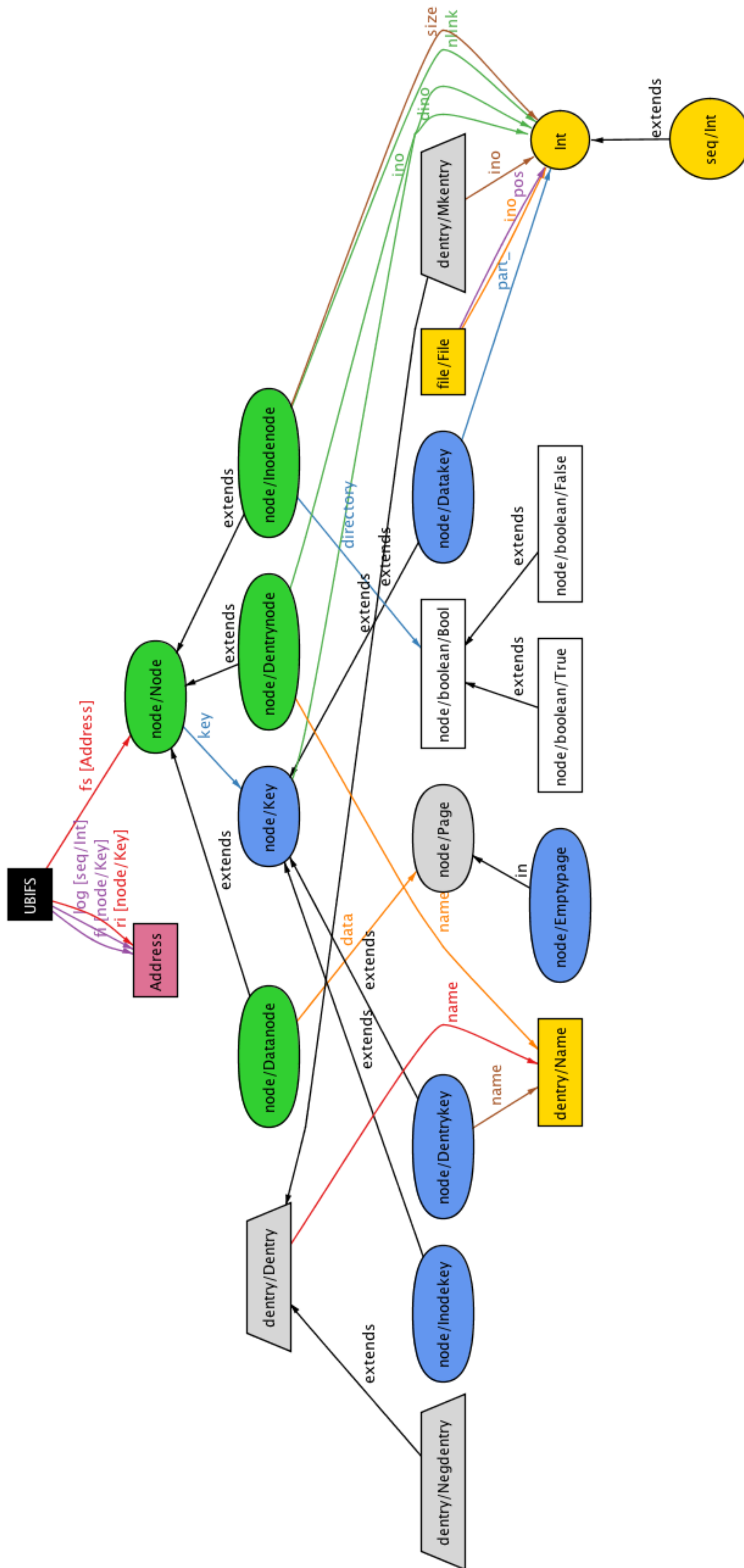
A.2.24 Assertion log_cons_unlink

```
assert log_cons_unlink {
  all u, u' : UBIFS, p_ino : Int, dent, dent' : Dentry |
       datanode_cons [u·fs, u·ri] and
       store_cons [u·fs, u·ri, u·log] and
       valid_dentry_with_parent [p_ino, dent, u·fs, u·ri] and
       valid_file_ino [dent·ino, u·fs, u·ri] and
       log_cons [u, u'] and
       UNLINK [p_ino, dent, dent', u, u'] ⇒ log_cons[u, u']
}
```

A.2.25 Fact Notnegative

```
fact Notnegative {
  all n : Node | n·size ≥ 0 and n·nlink ≥ 0 and n·dino ≥ 0
  all i : Int, k : Key | i in k·part_⇒  i ≥ 0
}
```

A.2.26 Metamodel of the UBIFS File System


UBIFS
Address
fs [Address]
log [seq/Int]
fi [node/Key]
ri [node/Key]
node/Node
key
extends
node/Key
node/Inodenode
node/Dentrynode
node/Datanode
data
node/Page
in
node/Emptypage
node/Datakey
node/Dentrykey
node/Inodekey
dentry/Dentry
dentry/Negdentry
dentry/Mkentry
dentry/Name
name
file/File
Int
seq/Int
size
nlink
dino
ino
pos
part_
directory
node/boolean/Bool
node/boolean/True
node/boolean/False