Under consideration for publication in Formal Aspects of Computing

Variations on an Alloy-centric Tool-Chain in Verifying a Journaled File System Model

Miguel A. Ferreira[1] and José N. Oliveira[2]

[1]Software Improvement Group, The Netherlands — m.ferreira@sig.eu
[2]CCTC, Universidade do Minho, Portugal — jno@di.uminho.pt

Abstract. Tool interoperability is among the main goals of the international Grand Challenge initiative. In the context of the Verifiable File System mini-challenge put forward by Joshi and Holzmann, our work has been focused on the integration of different formal methods and tools in a tool-chain for modelling and verification.

The current paper shows how to adapt such a tool-chain to the task in hands, aiming at reducing tool integration costs. The refinement of an abstract file store model into a *journaled* (flash) data model catering for wear leveling and recovery from power loss is taken as case study. This shows that refinement steps can be carried out within a shorter, reduced life-cycle where model checking in Alloy goes hand in hand with manual proofs carried out in the (pointfree) algebra of binary relations. This provides ample evidence of the positive impact of Alloy's lemma "everything is a relation" on software verification, in particular in carrying out induction-free proofs about data structures such as finite maps and lists.

Keywords: Grand challenge, formal methods, relational algebra, model checking

1. Introduction

There is a healthy trend in formal methods research driven by the idea of a *Grand Challenge* (GC). Hoare [23] revisited an old challenge in computer science: a verifying compiler, capable of performing extended static analysis of the programs it compiles. His paper defines a set of criteria for an international effort to drive research in computer science towards automatic software verification. As a response to the challenge made in [23] the *"Verified Software: Theories, Tools, Experiments"* conference was created, counting already two successful instances. In the first one (*VSTTE'05* [36]), Hoare *et al* [24] proposed that, since the conditions set in [23] were met, the time to start such a long term international research project had arrived. The project would focus on exploratory pilot projects to set the initiative in motion.

Correspondence and offprint requests to: J.N. Oliveira and M.A Ferreira

The GC initiative is expected to *"deliver a comprehensive and unified theory of programming"*, to *"prototype for a comprehensive and integrated suite of programming tools"* and to *"deliver a repository of verified software"* [24, Section 2]. Mondex [21] was the first GC pilot project. Later, the Verifiable File System (VFS) mini-challenge was put forward by Rajeev Joshi and Gerard Holzmann [28].

In the context of the VFS mini-challenge, the current paper builds upon two streams of previous research, one focused on the integration of both programming and logical tools that aid in the verification of formally specified operations regarding *satisfiability* proofs [10, 9, 12, 11]; and the other focused on performing such proofs by calculation [40], taking advantage of binary relation algebra [1]. In the former, we have proposed to combine different formal specification languages and make their tool-sets interoperate, so as to form a *tool-chain* supporting a development and verification life cycle that yields checked designs. Such a tool-chain should fulfill the following requirements:

- promote incremental development and verification of specifications;
- be agile enough to encourage users to verify even the smallest unit of their specifications;
- be capable of producing immediate feedback on the problems unveiled;
- be capable of performing fully automated proofs;
- be amenable to automatic code generation.

All requirements above, except for full automation of proofs, were achieved and reported in [11]. The contribution of the current paper is to build upon such results in two different (but related) directions:

- cater for refinement proofs;
- show how judicious use of relational pointfree notation [1] enables swift proofs of correction, even for complex refinements.

In this context, and further to the VFS verification exercise already reported in [11], we have selected for the current paper to report on the refinement of

- the file store model, so as to include a *journaling* mechanism that allows for higher performance and reliability in face of power loss; the inspiration for such a refinement step comes from work by Schierl *et al* [45] on the formalization of UBIFS file system for flash memory;
- the *delete* operation (specified and verified in [9]) downto the journaled level.

As will be explained in the sequel, refinement steps don't call for the whole machinery of the original tool-chain [11] and can be carried out within a shorter, or reduced life-cycle. This will be referred to as "Alloy centric" due to the central role of Alloy [26] (which was already noticeable in the complete tool-chain) in modeling and checking design decisions. Alloy helps a great deal to detect naive, subtle or rare corner case mistakes. Mistakes of this kind often go undetected by test cases, and are known to jam theorem provers for unforeseeable reasons.

Paper structure. Section 2 presents the motivation for building a tool-chain for software modelling and verification, and introduces the tools it is composed of. Section 3 address the integration of languages and respective tools, aiming at an agile tool-chain design capable of discharging both satisfiability and refinement proofs. Section 4 introduces the paper's case study: a relational model of a journaled file store. The link from the abstract file store to the refined (journaled) file store is presented in Section 5, including data type invariants and the operation that deletes elements of the file store. Section 6 gives the last touch on the journaling mechanism by introducing in-device caching to increase fault recovery performance. The advantages and limitations of the proposed tool-chain and strategy are addressed in Section 7. Sections 8 and 9 close the paper with, respectively, the authors review of related and future work. Standard facts of relation algebra and proofs of side-stream results are deferred to appendix A. We assume our target audience to be already using formal specification and verification techniques.

2. Tool-chain

The main motivation for the proposed tool-chain is to combine formal method tools for model checking, theorem proving, model animation, etc, in a way such that each tool is placed in the "right" step of the

proposed life-cycle. The version of the tool-chain which has been the subject of our experimentation [11] involves the following languages and tools.

Relational algebra. Following Tarski's formalization of *set theory without variables* [48], relation algebra has emerged as a language for expressing and reasoning about logical formulae in a concise, pointfree (PF) way. References [39, 40] show how to reason about data models using such PF-notation, in a typed way supported by type diagrams. The current paper exploits the same approach by regarding PF-notation and diagrams as the starting point of a proposed refinement step.

Alloy. This is a lightweight modelling language for software design inspired by Z [47] and developed by the Software Design Group at MIT [26]. Its foundations are first order logic and relation algebra. Alloy's lemma "everything is a relation" makes this highly declarative language simple and well integrated with relational PF-notation, as will be illustrated later.

Alloy's tool support is provided by the Alloy Analyzer intended for both development and verification of abstract models. This tool is capable of performing simulation as well as exhaustive verification in search for counterexamples to a given property. This tool relies on the Kodkod [49] SAT-based model finder that is able to perform a fully automated analysis over Boolean logic. To take advantage of Kodkod (or other SAT solvers) the Alloy Analyzer translates relational logic into Boolean logic and then applies the solver to find counterexamples. Whenever a counterexample is found, the corresponding Boolean formula is translated back to relational logic and displayed as an interactive diagram for user inspection.

These characteristics make Alloy a suited candidate for integration with relational PF-notation, since their similarities make the translation from one to the other almost direct [11].

VDM. The Vienna Development Method [2] is a mature formal method whose origins go back to the IBM Vienna Laboratory in the 1970s. The use of VDM associated languages to specify and guide the development of software has been widely described in the literature [13, 14]. VDM++ [43] is a widespread VDM dialect which, compared to ISO standard VDM-SL [44], introduces object oriented and concurrency features in the language. Tool support is one of the key strengths of VDM in general. From the wide variety of tools available we single out the Overture [31] Automatic Proof System (APS) [50] and the VDMTools [15] for type checking, interpretation and code generation.

HOL. This theorem prover [19, 46] (a descendant of the LCF theorem prover) was developed with hardware verification in mind. It is an interactive proof assistant designed for higher order logic, with a vast set of ready to use theories and proof tactics. Its function definition mechanism provides termination proofs for recursive functions for free.

The choice of the HOL [20] theorem prover was not explicitly ours, instead it was conditioned by the fact that a translation tool is available to convert VDM++ into HOL notation. The latest version (HOL4) [46] of the theorem prover is implemented on top of the Moscow ML interpreter, which runs the functional language Standard ML. HOL proof tactics are functions in the Moscow ML domain that, when applied to a HOL term, may produce a proof. Together with tacticals it enables the construction of specialized tactics for a given application domain. The Overture APS exploits this flexibility in defining VDM++ specific proof tactics capable of dealing with proof obligations arising from specifications.

3. "All-in-one" strategy

To effectively build a tool-chain it is necessary to have a strategy for each component as well as for the overall set of tools. The main goal of the strategy is to provide better verification techniques for better development of software.

"Better" development means that the first steps in specifying a given problem should be taken at the highest abstraction level possible, capturing the key aspects of the artifact under specification. This should be followed by incremental refinement of the specification, in order to obtain an executable version that can be used to validate functional requirements with stakeholders. Once verified, the executable specification is translated to source code in some mainstream programming language. The leap from abstract specification to executable specification must allow for early detection of failing functional requirements.

"Better" verification means that before tackling full-fledged proofs, confidence in the specification should

be gained. In this way, one avoids attempting proofs that could be demonstrated impossible by counterexamples, or that add no value to the development since they fail the user requirements.

Two kinds of proof are considered in the remainder of this paper. One is known as *satisfiability* [27]: for every operation Op whose input is of type A and whose output is of type B, proof obligation (PO)

$$\forall\ a\ \cdot\ a \in A \wedge \text{pre-}Op(a) \Rightarrow \exists\ b\ \cdot\ b \in B \wedge \text{post-}Op(b, a) \tag{1}$$

should be discharged. Because $a \in A$ and $b \in B$ check for the data type invariants associated to A and B, respectively, this PO is also referred to as *invariant preservation* [27]. In case Op is deterministic and performed uniformly over a state space ($A = B$), PO (1) shrinks to

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

The second kind of proof has to do with refinement steps in which an operation such as Op above is implemented by another operation Op' running on a strengthened, or more detailed, state space A' [35], glued to the abstract space A by an *abstraction invariant* (ai): for all suitably typed x, x', y, y',

$$ai(x, y) \wedge \text{post-}Op'(y', y) \Rightarrow \exists\ x'\ \cdot\ \text{post-}Op(x', x) \wedge ai(x', y') \tag{2}$$

should hold. Typically, $ai(x, y)$ will be decomposed in two parts: an abstraction function $x = af(y)$ and a concrete invariant $ci(y)$ [37] at low-level. In this case, which covers the examples given in this paper, (2) instantiates to

$$ci(y) \wedge \text{post-}Op'(y', y) \Rightarrow \text{post-}Op(af(y'), af(y)) \wedge ci(y') \tag{3}$$

Refinement steps involve other, complementary proofs (related, for instance, to deadlock freedom) which are not listed because they will not be considered in the examples given later in the current paper. (See [35] for details.)

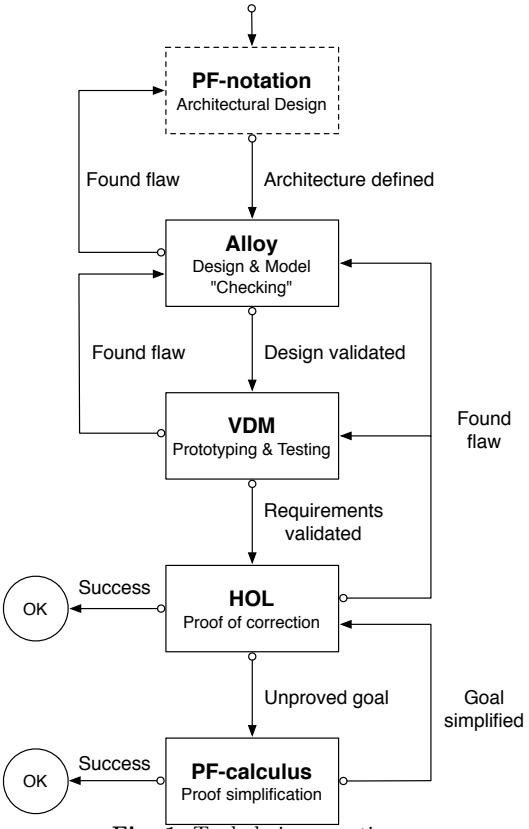


Fig. 1: Tool-chain operation.

3.1. Satisfiability proofs

Concerning satisfiability proofs (1), the following situations can take place:

1. While specifying the overall architecture of a system, several interests are at stake. Often these interests are contradictory. A well founded notation which is paradigm-, platform- and technology-independent is welcome to enable reasoning about the high-level design.
2. During the design phase, several experiments are performed to assess different design options for each operation Op. A *model checker* able to automatically generate counterexamples to (1) and thus suggest how to improve Op is welcome.
3. Op satisfies PO (1) but is semantically wrong, for it ends up not behaving according to the requirements. To prevent this situation, running the model as a *prototype* subject to a test suite in an interpreter is welcome.
4. Both the model checker and the test suite above do not find any flaws. In this case, a *theorem prover* is welcome to mechanically check (1).
5. PO (1) is too complex for the theorem prover. In this situation, the ultimate hope is a pen-and-paper manual proof, or some kind of exercise able to decompose such a complex PO into smaller sub-proofs.

This 5-step design scenario calls for a PO discharge strategy based on, respectively:

1. A highly abstract mathematical notation providing for agile sketching of the data objects of interest, their relationships, constraints and mutual dependencies, and for algebraic manipulation — we have chosen relational algebra notation [1] in the style proposed in [40] [1].
2. A model checker for timely generation of uninterpreted, unexpected counterexamples — we have chosen Alloy for this purpose.
3. An interpreter enabling one to carry out semantically meaningful animation and testing — for this purpose we have chosen the VDMTools.
4. A theorem prover, HOL in our case, thanks to the Overture proof system.
5. A pen-and-paper proof strategy regarding POs as "mathematical objects" which can be calculated upon. For this stage we have been using the PO calculus described in [40], where POs are represented by arrows which can be put together or decomposed into simpler ones, as already illustrated in [11].

This "all-in-one" strategy is depicted in Figure 1. The process starts from a highly abstract model of the architectural design of the target system, either in relational pointfree notation or directly in Alloy. Note the dashed line of the topmost box in Figure 1 (PF-notation), meaning that it is an optional stage. Although Alloy is not able to prove properties, it is very useful in finding counterexamples spotting where and why properties fail.

After validating the design in Alloy, the model is translated to VDM++, where more detail is introduced. (Due to Alloy's notational compactness, the equivalent VDM++ specification becomes more verbose.) In the VDM++ stage it is already possible to validate all functional requirements, since the specification becomes executable. Validation can be carried out through unit tests [14, Section 9.5], combinatorial tests [32], or by interpreting (animating) the specification. Should dynamic analysis performed at VDM++ level detect any design flaw, the process goes back to the Alloy stage to suppress defective behaviour. Once the specification looks adequate and captures all functional requirements, the Overture APS is used to generate all the POs arising from the VDM++ model and piping these to HOL for mechanical discharge.

The last stage (pen-and-paper proof) caters for POs which HOL could not prove and Alloy could not refute: the worst scenario. The idea is to use PF-calculation at this stage, aiming at simplifying POs e.g. by splitting them into smaller goals, which are fed back to HOL, and so on and so forth.

This is the strategy adopted in [10, 9, 11] for discharging satisfiability POs arising in an abstract model of a flash file system, whose refinement is the main subject of the current paper.

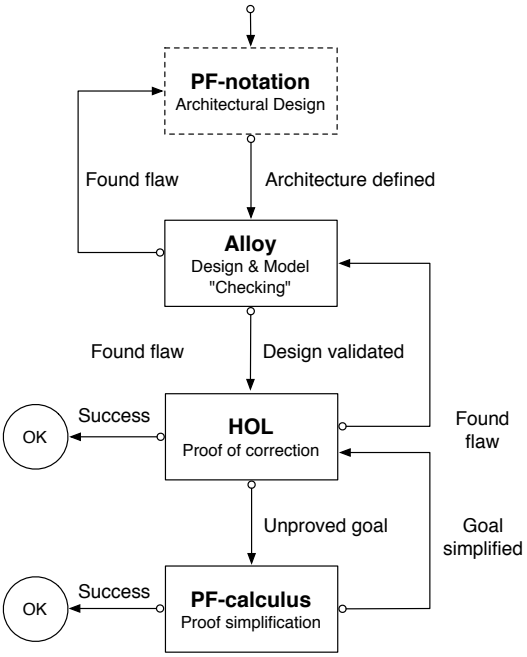


Fig. 2: Simplified tool-chain.

3.2. Refinement proofs

Although the complete tool-chain shown in the previous section could be used (as is) for refinement proofs, it would over-complicate the exercise. The explanation for this simplification follows from the fact that in refinement steps there are no (subjective) user requirements to be validated, thus rendering execution and

[1] In a sense, this notation plays the role UML normally does in informal software design. The advantages of its also being very amenable to calculation are obvious.

animation of the model unnecessary. In other words, refinement requirements are all stable (as captured by the underlying refinement theory [35]), directly leading to mathematical proofs.

This means that that the stage of prototyping and testing in VDM++ can be removed from the tool-chain without harm. Figure 2 depicts a simplified version of the tool-chain of Figure 1, where the VDM++ stage was removed. Thus the workflow passes directly from the Alloy stage, after model-checking, to the HOL stage for the full fledged proof.

This simplification presents problems with respect to sewing Alloy to HOL. In Figure 1, the translation from Alloy to VDM++ was carried out following a set of rules that had been previously identified [9, 10, 11]. These rules make it quite simple to translate Alloy models to VDM++, apart from operational issues arising from the fact that the target VDM++ is intended for execution. However, from Alloy to HOL there are no preset rules to guide the translation, let alone an automatic translator. We envisage two ways out of this problem: (1) study a systematic way of translating Alloy to HOL; and (2) remove HOL from the tool-chain as well. Let us now briefly analyze the cost benefit relation of each of these possibilities.

On the one hand, the development of a set of Alloy to HOL semantics-preserving translation rules would enable proof discharge in HOL. On the other hand, stepping over the HOL stage and going directly into manual PF-calculation would be practically viable should the Alloy model in hands be close enough to the PF-notation itself. There are two main ingredients in this alternative: (a) PF-calculation is less error-prone because of its compact notation and calculus; (b) difficult steps in proofs (often resorting to lemmas) are model-checked in Alloy prior to the actual calculation, thus saving the effort of trying to prove invalid steps.

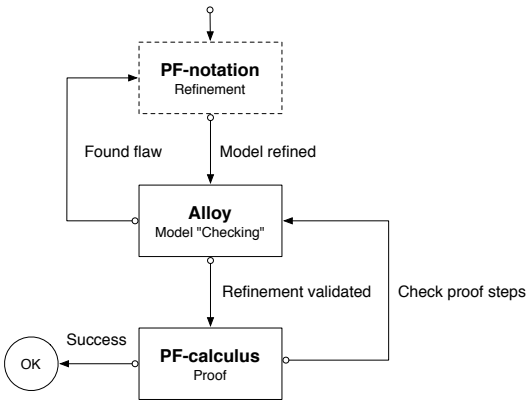


Fig. 3: Simplified tool-chain.

The remainder of this paper will provide ample illustration of the advantages of putting ingredients (a) and (b) together using the shortest version of the tool-chain (no VDM++, no HOL) depicted in Figure 3, which will be shown to be enough to formulate and prove data refinement development stages.

Let us see this process in more detail. According to (2,3), to prove the intended refinement it is necessary to show that once running a refined operation (Op') in a state reachable from a given abstract state (x), there exists another such state (x') which abstracts the after-state of Op'. Moreover, any new operation invented at low level should remain "invisible" at high level, that is, it should refine $Skip(x', x) \triangleq x' = x$. In the case of (3), this boils down to showing that such low-level operations respect (ie. preserve) low-level invariants.

Following Figure 3, once the refined model is conjectured and encoded in Alloy, refinement POs are model checked using the Alloy Analyzer. Absence of counterexamples leads to the calculation stage. Lemmas introduced in proofs (or particularly difficult proof steps) are also model-checked. The process is not finished until all manual calculations are over.

We proceed to the presentation of our case study recalling that, in the problem in hands, satisfiability at the abstract level was previously proven for the *delete* [9, 40] and *open* [11] operations.

4. Relational model of a journaled (flash) file system

In a typical file system, file store's data and meta-data are stored in the device that hosts the file system. In order to increase performance, some meta-data are stored in central memory (RAM) dramatically decreasing the update latency. However, this approach leads to consistency problems when, for some unexpected reason, the device is removed, or the whole system crashes. In such faulty situations, meta-data stored in central memory are lost, and their counterpart stored in the device will most likely be out of date.

One way to add robustness to the file system against faults of this kind is to store in the device a *backlog* of the operations that have been performed. Such a backlog is often referred to as a *journal* [45]. Altogether, journal and remaining meta-data stored on the device should make it possible to rebuild the meta-data that were stored in central memory as it was before the fault happened.

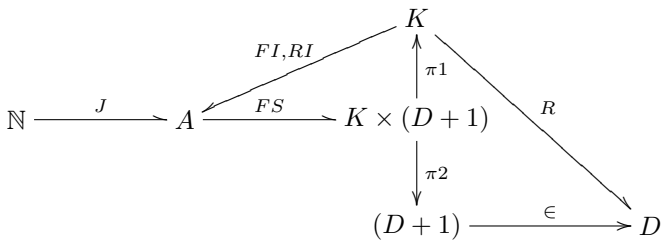
While journaling and de-centralized meta-data improve file system performance, they also add to the

complexity of the file store model and its invariant. Dealing with this added complexity is the goal of the model presented in the sequel.

Refined file store. Our starting point is the abstract relational model of a file system presented in [40, 11]. From this model we single out the file store (data structure that holds files) which is modeled by a finite and univocal (simple) relation (a mapping) between paths and files, as shown aside. Note the use of "hooked arrows" to depict this kind of relation already adopted in [40, 11]. This file store, hereinafter referred to as the *abstract file store*, is depicted as relation R in diagram (4) below, where we also abstract $File$ by D (for data) and $Path$ by K (for key, since paths are unique). In so doing we implicitly show that the refinement step put forward in this section is *generic*, that is, it does not depend on any specific details of the previous model concerning the $Path$ and $File$ datatypes.

$$Path \xrightarrow{fileStore\ s} File$$

The journaled refinement of such an abstract file store, hereinafter referred to as *journaled file store*, is a collection of four data structures, labelled FI, J, RI and FS in the diagram which follows, where letters F, I, J, R and S stand consistently for *flash*, *index*, *journal*, *RAM* and *store*, respectively:



(4)

In detail:

- FS (flash store) is stored in the device and holds the actual data, where A is the datatype of flash memory *addresses* and "maybe type" $D+1$ accommodates both valid data (D) and the DEL mark (1) intended for recording data deletion, as we shall soon see. Type $K \times (D+1)$ is inhabited by pairs (k, x) of keys and such *maybe data* values. Projections π_1 and π_2 are such that $\pi_1(k, x) = k$ and $\pi_2(k, x) = x$. Relation $\in$ is the membership of type $D+1$: $d \in x$ holds iff x does not keep a DEL mark and d is its contents.
- The structures represented by FI (flash index) and J (journal) hold in-device meta-data as well, whereas RI (RAM index) keeps similar data in central memory.
- RI provides for fast indexing, mapping each key in K to the address that currently holds its data in FS; so, by chaining RI and FS one should be able to rebuild the information kept in R. But there is some redundancy in the refinement, as FS also keeps the converse relationship between addresses (A) and keys (K) — a redundancy intended for power loss recovery, as we shall soon see.
- Under normal operation, device removal is preceded by a *commit* operation whereby the contents of RI are saved in FI.
- Abnormal operation (power loss or abrupt device removal) calls for the help of journal J, which keeps the list of addresses which have been created since the last commit operation. This means that, on such situations, a *replay* process should take place able to rebuild RI (volatile) from the in-device (FS, FI, J) triple.
- Finally note the fact that, in diagram (4), list J is modelled as an association of natural numbers ($\mathbb{N}$) to addresses (A) indicating the position of each address in the list.

These four structures of the refined state space are suggestively recorded in Figure 4, extracted verbatim from reference [45]. This picture and paper eloquently address the strategy of *wear leveling* and *power loss* recovery. Note for instance how deletion of entry $KEY4$, stored in address 1, means adding a new entry to FS (address 9) where $KEY4$ is marked as deleted (DEL). In other words, to delete n cells in FS one must have n extra free cells. Should this not be the case, a garbage collection operation has to take place, cleaning (evenly) all redundant entries such as $KEY4$ in address 1. Lack of space persisting, the device will be full.

Alloy encoding. Prior to the formalization of this journaling refinement step, let us see how the relations of diagram (4) are encoded as Alloy signatures, intended for model checking. While abstract states (AS) are modeled as Alloy maps from keys (K) to data (D),

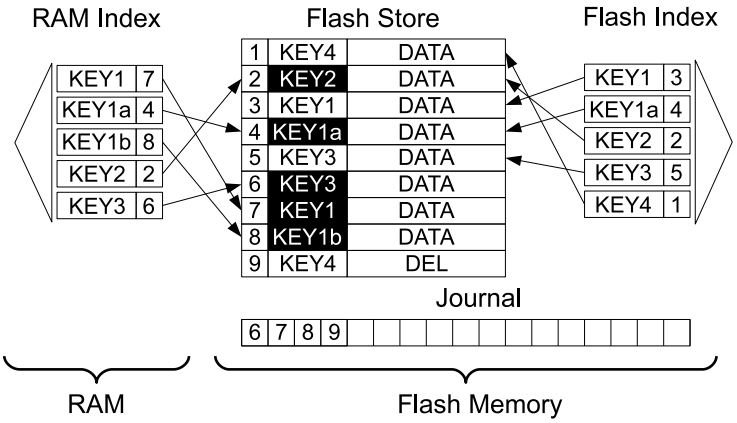


Fig. 4. Journaled file store data structures (picture quoted from [45]).

```
sig AS { r : K → lone D }
```

the concrete ones are captured by the four-entry signature,

```
sig CS {
  j : N → lone A,
  fs : A → lone Entry,
  fi : K → lone A,
  ri : K → lone A
}
```

where entries in the flash store are pairs

```
sig Entry { key : one K, value : one DataCell }
```

whose right element, $D + 1$ in diagram (4), is modeled by an abstract signature extended by DEL marks and proper data:

```
abstract sig DataCell {}
one sig Del extends DataCell {}
sig Data extends DataCell { data: one D }
```

5. Refinement process

Abstraction invariant. We start by establishing the abstraction invariant (ai) gluing abstract states — R in diagram (4) — to concrete states — the 4-tuple (FS, RI, FI, J) in the same diagram. As anticipated by refinement equation (3), we partition ai in terms of an abstraction function af and concrete invariant ci.

Let $R \cdot S$ denote the composition of relations R and S defined in the usual way: $b(R \cdot S)c$ holds wherever there exist one or more mediating a such that bRa and aSc both hold. Relying on relational composition alone, we define af as follows,

$$af(FS, FI, RI, J) \triangleq (active\ FS) \cdot RI \tag{5}$$

where relation

$$A \xrightarrow{active\ FS} D \triangleq \in \cdot \pi_2 \cdot FS \tag{6}$$

is easy to grasp by chasing the arrows in diagram (4). It captures the A to D relationship recorded in FS, thus encompassing all addresses in FS which do not lead to DEL entries [2]. Chaining this with RI yields the K to D map stored in the device. The Alloy encoding of definitions (5) and (6) is as follows, where *cs* (CS) stand for *concrete state*:

```
fun af[cs: CS] : K → D { (cs·ri)·(cs·fs·active) }

fun active[x: CS·fs] : A → D { x·value·data }
```

(Note that reverse order in which Alloy chains the arguments of relational composition.)

Recall that the abstract state is inhabited by *simple* relations, that is, relations which are univocal: no two different data items are associated to the same key. Relational simplicity is a property which plays a major role in data modelling and is easy to establish: S is said to be *simple* iff $S \cdot S^\circ \subseteq id$ holds, where id is the identity function (ie., equality relation) and S° denotes the *converse* of S, that is, the relation such that $a(S^\circ)b$ holds iff bSa holds. (See [40] for details of a taxonomy of binary relations established in this way.)

So, for af to be properly typed, it must always deliver one such relation R. Since composition preserves simplicity, *active* FS will be simple once FS is so, since π_1 (a function) and $\in$ (the converse of an injection, see e.g. [41]) are by definition simple. In the same vein, $(active\ FS) \cdot RI$ will be simple because RI is simple. Note how elementary properties of relation algebra help in type checking the diagram "on the fly".

Concrete invariant. We have seen above that simplicity of RI and FS are the first requirements of concrete invariant *ci* to consider. To save us from having to write these explicitly, we record them in the type diagram (4) by using "hooked arrows". Thus J and FI are meant to be simple relations too.

5.1. Operation refinement

Let us proceed to the refinement process itself, whereby abstract operations over abstract states of type $K \longrightarrow\!\!\!\!\!\!\sim\ D$ are implemented at low level, running on top of concrete states whose type is depicted in diagram (4). In the particular flash file store context, this means abstract operations such as creating a new file, modifying an existing file, deleting a file, and so on [9].

The delete operation. Among these, we single out the operation of file deletion for two reasons. First, its abstract specification has been dealt with in [40], already in the pointfree relational style. Second, because its implementation is far more elaborate than first thought due to the *wear levelling* and *power-loss recovery* non-functional requirements. Recall from [40] that file deletion was specified as follows,

$$\text{post-}FS\_DeleteFileDir\_FStore(S, R', R) \triangleq R' = R \cdot \Phi_{(\notin S)} \tag{7}$$

where argument set S tells which paths (keys in our generic model) are to be deleted, and $\Phi_{(\notin S)}$ is a *filter* — the *coreflexive* relation associated to predicate $x \notin S$ [3]. The Alloy counterpart of (7) reads as follows (abbreviating the long identifier for economy of notation):

```
pred abs_Delete[r,r': AS·s, s: set K] { r' = (K·crflxv − s·crflxv)·r }
```

where function $crflxv$ represents sets as coreflexives:

```
fun crflxv[s: set univ] : univ → univ { iden & (s → s) }
```

By introducing variables in the after state R' of post-condition (7), we confirm the intended "domain subtract" semantics of the operation:

$d\ R'\ k$ holds iff $d\ R\ k \wedge k \notin S$

[2] So, only RI and FS contain relevant information: J and FI will remain as implementation details.

[3] In general, given a predicate p, coreflexive relation Φ_p is such that $b\ \Phi_p\ a$ holds iff $(b = a) \wedge (p\ a)$; that is, Φ_p is the relation that maps every a which satisfies p (and only such a) onto itself. Clearly, such a relation is a fragment of the identity relation ($\Phi_p \subseteq id$).

How do we implement this operation at journaled, flash level? At first sight, performing a similar domain-subtract operation on the RAM index RI,

$$RI' = RI \cdot \Phi_{(\notin S)} \tag{8}$$

and leaving FS, FI and J unchanged would do, since the smaller RI the smaller the abstract state delivered by af (5). And it does in fact, as the following instance of refinement PO (3) and reasoning shows [4]:

$$RI' = RI \cdot \Phi_{(\notin S)} \wedge FS' = FS \quad \Rightarrow \quad af(FS', RI') = af(FS, RI) \cdot \Phi_{(\notin S)} \tag{9}$$

$\Leftrightarrow$ { equal by equal substitution }

$af(FS, RI \cdot \Phi_{(\notin S)}) = af(FS, RI) \cdot \Phi_{(\notin S)}$

$\Leftrightarrow$ { (5) }

$(active\ FS) \cdot (RI \cdot \Phi_{(\notin S)}) = ((active\ FS) \cdot RI) \cdot \Phi_{(\notin S)}$

$\Leftrightarrow$ { composition is associative }

$true$

Why aren't things so easy in practice? Note that, for every key deleted in RI there is an address in FS which becomes available for further writing. And further delete/write cycles may overwrite such an address over and over again, thus contradicting *wear leveling*, a non-functional requirement intended in general to prolong the service life of erasable computer storage media.

On the other hand, in the event of a power loss RI will be lost, for it lives on volatile RAM. This could in part be remedied by exploiting the redundancy of FS (4) and running an operation able to recover the K to A association it keeps,

$$K \xrightarrow{index\ FS} A \quad \triangleq \quad (\pi_1 \cdot FS)^\circ \tag{10}$$

— that is,

```
fun index[x: CS·fs] : K → A { ~(x·key) }
```

in Alloy — provided the following *consistency* clause is added to the concrete invariant:

$$RI \subseteq index\ FS \tag{11}$$

Note, however, that (11) cannot be strengthened to an equality, for $index\ FS$ is not simple in general: by construction, it keeps track of all addresses which have been used to record data for a given k. Besides, information is missing about which addresses correspond to the most recent updates. So RI is not recoverable at all.

This leads to a revision of the model which brings journal J into the scene, keeping the order in which addresses have been created.

5.2. Revised refinement step

The revision consists, first of all, in adding (11) to the concrete invariant and ensuring referential integrity — addresses logged in J are exactly those found in FS:

$$ci(FS, FI, RI, J) \ \triangleq \ RI \subseteq index\ FS \ \wedge \ \rho J = \delta FS \tag{12}$$

The second conjunct of clause (12) resorts to the ρ and δ relational operators [40], respectively yielding the *range* and *domain* of a relation, expressed as coreflexive relations.

A consequence of (12) worthwhile mentioning is the *injectivity* of RI. This arises from the simplicity of

[4] Since FI and J don't play any role in the model so far, they have been temporarily omitted from the argument list of af, for improved readability.

$\pi_1 \cdot FS$ and the two rules of thumb: *converse of simple (resp. injective) is injective (resp. simple)* and *smaller than injective (resp. simple) is injective (resp. simple)* [40]. Moreover,

$$\rho(index\ FS) = \delta FS \tag{13}$$

as can be easily checked using the algebra of domain and range — see e.g. (65) and (72) in the appendix. (Also note that a function f is a totally defined relation, that is, $\delta f = id$.)

Finally, a third clause has to be added to *ci* rendering RI functionally dependent on the other components of the concrete state,

$$RI = replay(FS, FI, J) \tag{14}$$

thus ensuring that, at any time, RI can be rebuilt from persistent flash data by running function *replay*. The specification of this most important ingredient of the revised model follows.

The *replay* function. The semantics of *replay* are based on handling *index* FS (10), in two steps:

- first, for each key in *index* FS select the address which holds its most recent update (this will yield a simple relation from K to A);
- second, filter deleted keys out.

These steps are made formal in the following definition of *replay*

$$replay(FS, FI, J) \triangleq (active\ FS) \lhd (index\ FS \upharpoonright (\geq_J)) \tag{15}$$

where $\geq_J$ is the abbreviation of

$$A \xleftarrow{\geq_J} A \quad \triangleq \quad J \cdot \geq \cdot J^\circ \tag{16}$$

and the two binary relational combinators $\lhd$ and $\upharpoonright$ are explained next [5].

The $\lhd$ combinator. This combinator, defined by

$$S \lhd R \triangleq \delta S \cdot R \tag{17}$$

(read $S \lhd R$ as "R if S is defined") post-restricts a given relation R by the domain of some other relation S. Clearly, properties

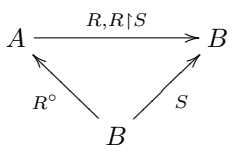
$$(S \lhd R) \cdot T = S \lhd (R \cdot T) \tag{18}$$

$$S \lhd (R \cup V) = (S \lhd R) \cup (S \lhd V) \tag{19}$$

hold. Combinator (17) is used in (15) in the second step, filtering deleted keys out (ie. those not in *active* FS). Its encoding in Alloy is as follows:

```
fun ifdef[s,r : univ → univ] : univ → univ { r·(s·dom)·crflxv }
```

The *thinning* combinator. The aim of the *thinning* combinator ($\upharpoonright$) in the current context is to convert a given relation R into a smaller, simple relation by looking at particular (eg. maximal) elements of its range relative to some ordering. This is captured by the diagram aside and universal property



$$X \subseteq R \upharpoonright S \quad \Leftrightarrow \quad X \subseteq R \ \wedge \ X \cdot R^\circ \subseteq S \tag{20}$$

which ensures that $R \upharpoonright S$ is the largest sub-relation X of R such that, for all $b', b \in B$, if there exists $a \in A$ such that $b'Xa \wedge bRa$, then $b'Sb$ holds [6]. This pointwise meaning is captured by the following encoding in Alloy:

```
fun thinby[r: K → A, s: A → A] : K → A { { a : r·dom, b : a·r | all b' : a·r | b' in s·b } }
```

[5] A note on relational operator precedence, intended for saving parentheses: unary operators bind tighter than binary ones.
[6] This combinator is a generalization of the *max* operator of [1].

Let, for instance, S be id (the equality relation) and R be simple in (20). Then, since $R \cdot R^\circ \subseteq id$ holds, the maximal solution of (20) is $X := R$. That is, $R \upharpoonright id = R$ for R simple. In case R is not simple, $R \upharpoonright id$ will be the largest deterministic fragment of R. Among the properties of this combinator, we single out the following,

$$(R \upharpoonright S) \cdot \Phi = (R \cdot \Phi) \upharpoonright S \tag{21}$$

$$R \upharpoonright S = R \upharpoonright (\rho R \cdot S \cdot \rho R) \tag{22}$$

$$R \upharpoonright S \text{ is simple} \Leftarrow S \text{ is antisymmetric} \tag{23}$$

$$(R \cup S) \upharpoonright U = (R \upharpoonright U) \cap U/S^\circ \cup (S \upharpoonright U) \cap U/R^\circ \tag{24}$$

as well as the following corollary of (24),

$$(R \cup S) \upharpoonright U = (R \upharpoonright U) \cup (S \upharpoonright U) \;\Leftarrow\; R \cdot S^\circ \subseteq \bot \tag{25}$$

all relevant in the sequel. For instance, thanks to (23), the *thinning* combinator *"simplifies" index* FS in (15) via relation (16), which orders addresses by comparing their relative positions in J, larger positions meaning more recent updates:

$$a \geq_J b \;\Leftrightarrow\; \exists\, i, j \;\cdot\; aJi \wedge bJj \wedge i \geq j$$

However, to ensure (16) antisymmetric, simple J needs also to be injective, meaning no address duplication in journal J:

$$\geq_J \cap \geq_J{}^\circ \subseteq id \;\Leftarrow\; J \text{ simple and injective} \tag{26}$$

(See proof in appendix A.)

Final touch. From definition (15) one immediately sees that clause (14) strengthens (11), since both $B \upharpoonright S$ and $S \triangleleft R$ are subrelations of R, in general. So (14) replaces (11) in the final version of concrete invariant ci, which also records the injectivity of J:

$$ci(FS, FI, RI, J) \;\triangleq\; RI = replay(FS, FI, J) \;\wedge\; \rho J = \delta FS \;\wedge\; J \text{ is injective} \tag{27}$$

Encoded in Alloy, this invariant becomes

```
pred ci[cs : CS] { cs·ri = cs·replay and cs·j·ran = cs·fs·dom and injective[cs·j,A]}
```

where, last but not least, the *replay* function (15) is given by:

```
fun replay[cs: CS] : K → A {
  let geq = ^(ordering/prev) + N·crflxv,
      geqj = ~(cs·j)·geq·(cs·j),
      ix = thinby[cs·fs·index, geqj] |
    ifdef[cs·fs·active,ix]
}
```

5.3. Revised delete operation

As happens with the *replay* function, the revised delete operation is better explained in its conceptual steps:

- first, it should be possible to assign the key of every entry to be deleted to a fresh store address where it will be marked as deleted;
- second, the journal must be updated accordingly;
- finally, the RAM index should be restricted as in the previous version (8), thus denying access to deleted entries.

Note the *partial* behavior of this operation when compared to its abstract counterpart: there may not exist enough fresh addresses for the operation to be completed. This is captured by the existential flavor of the corresponding post-condition, as follows:

Definition 5.1 (Post-condition of the delete operation at journaled level). There exist simple and injective $K \xrightarrow{N} A \xleftarrow{M} \mathbb{N}$ where N associates fresh file store addresses to keys to be deleted and M orders such addresses in an underspecified way, such that

$$J' = J \cup M \tag{28}$$
$$FS' = FS \cup del\ N \tag{29}$$
$$RI' = RI \cdot \Phi_{(\notin S)} \tag{30}$$
$$FI \text{ remains unchanged} \tag{31}$$

under the following conditions: (a) addresses in N are fresh

$$FS \cdot N = \bot \tag{32}$$

(where $\bot$ is the least relation of its type) and precisely those listed in M:

$$\rho N = \rho M \tag{33}$$

(b) keys in N are exactly those to be deleted:

$$\delta N = \Phi_{(\in S)} \tag{34}$$

(c) positions in M are such that $J \cup M$ appends M to J:

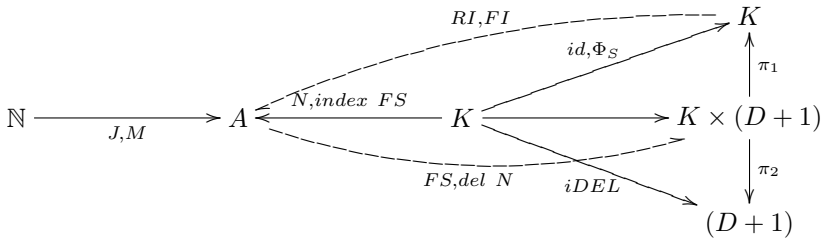
$$M^\circ \cdot \top \cdot J \subseteq > \tag{35}$$

(d) finally, $del\ N$ associates DEL marks to fresh addresses and their keys,

$$del\ N \triangleq \langle id, iDEL \rangle \cdot N^\circ \tag{36}$$

where the split combinator $\langle f, g \rangle$ pairs the results of its argument functions f and g [1] and $iDEL$ injects the DEL mark in $D + 1$ [7].
□

The diagram which follows helps in type checking the post-condition just given,



(37)

as well as the corresponding Alloy encoding:

```
pred Delete[cs,cs': CS, s: set K] {
  some n: K → lone A, m: N → lone A {
    injective[n, A] and injective[m, A]
    no n·(cs·fs) and n·ran = m·ran
    n·dom = s and cs·j·Top·(~m) in ^(ordering/prev)

    cs'·j = cs·j + m
    cs'·fs = cs·fs + n·del
    cs'·ri = (cs·fs·dom·crflxv − s·crflxv)·(cs·ri)
    cs'·fi = cs·fi
  }
}
```

[7] $iDEL$ abbreviates expression $i_2 \cdot !$, where i_2 is the right injection associated to sum $D+1$ and ! delivers the unique inhabitant of type 1. This notation is adopted from [1].

where

```
fun del[n: K → A] : A → Entry { { a: n·ran, e: Entry | e·key in n·dom and e·value = DEL } }
```

Finally, for diagram (37) to type-check we still need to prove that arrows FS' and J' are simple,

$$FS' = FS \cup del\ N \text{ is simple} \tag{38}$$
$$J' = J \cup M \text{ is simple} \tag{39}$$

as RI' is trivially so. The calculations of (38) and (39) can be found in appendix A.

A number of facts are useful recording at this point, for later use. Proofs of the ones left unjustified can also be found in appendix A:

- All fresh entries in FS' are inactive:

$$active(del\ N) = \bot \tag{40}$$

  This arises from the disjointness condition $\in \cdot iDEL = \bot$.
- $index$ recovers N from $del\ N$:

$$index(del\ N) = N \tag{41}$$

- J' adds nothing to the update of $index\ FS$:

$$index\ FS \upharpoonright (\geq_{J'}) = index\ FS \upharpoonright (\geq_J) \tag{42}$$

Checking the proposed refinement. It is handy to split proof obligation (3) in its two components, one dealing concrete invariant maintenance,

$$ci(y) \wedge \text{post-}Op'(y', y) \Rightarrow ci(y') \tag{43}$$

and the other ensuring safe refinement:

$$ci(y) \wedge \text{post-}Op'(y', y) \Rightarrow \text{post-}Op(af(y'), af(y)) \tag{44}$$

The instance of (44) for the *Delete* operation is, in Alloy syntax:

```
assert po44 {
  all cs,cs': CS, s : set K |
    (ci[cs] and Delete[cs,cs',s]) ⇒ abs_Delete[cs·af,cs'·af,s]
}
```

The instance of (43) for the situation in hand (27) splits into,

$$ci(FS, FI, RI, J) \wedge \text{clauses (28) to (36)} \Rightarrow \rho J' = \delta FS' \tag{45}$$
$$ci(FS, FI, RI, J) \wedge \text{clauses (28) to (36)} \Rightarrow J' \text{ is injective} \tag{46}$$
$$ci(FS, FI, RI, J) \wedge \text{clauses (28) to (36)} \Rightarrow RI' = replay(FS', FI', J') \tag{47}$$

giving rise to the Alloy assertions:

```
assert po45 {
  all cs,cs': CS, s : set K |
    (ci[cs] and Delete[cs,cs',s]) ⇒ cs'·j·ran = cs'·fs·dom
}

assert po46 {
  all cs,cs': CS, s : set K |
    (ci[cs] and Delete[cs,cs',s]) ⇒ injective[cs'·j,A]
}

assert po47 {
  all cs,cs': CS, s : set K |
    (ci[cs] and Delete[cs,cs',s]) ⇒ cs'·ri = cs'·replay
}
```

Calculation of (44) proceeds by re-writing the consequent of the implication, assuming the antecedent and substitutions implicit:

$$
\begin{array}{ll}
& af(FS', FI', RI', J') = af(FS, FI, RI, J) \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \quad \{ \ (5) \ \} \\
& (active\ FS') \cdot RI' = ((active\ FS) \cdot RI) \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \quad \{ \ (30)\ ;\ (29) \ \} \\
& active(FS \cup del\ N) \cdot RI' = (active\ FS) \cdot RI' \\
\Leftarrow & \quad \{ \ \text{Leibniz} \ \} \\
& active(FS \cup del\ N) = active\ FS \\
\Leftrightarrow & \quad \{ \ \textit{active}\ (6)\ \text{distributes through union} \ \} \\
& (active\ FS) \cup (active(del\ N)) = active\ FS \\
\Leftrightarrow & \quad \{ \ (40) \ \} \\
& active\ FS = active\ FS \\
\Box &
\end{array}
$$

Not so immediate as calculation (9) performed in the starting refinement step, PO (44) was easy to discharge, after all. So, the crux of the refinement step must reside elsewhere: in fact, in preserving the concrete invariant's conditions which ensure referential integrity and power loss recovery, as we shall see shortly.

The first two such POs to check, (45) and (46), are still easy exercises in relation algebra (see appendix A). By contrast, the calculation of PO (47) is by far the most expensive and complex of the whole refinement exercise, giving evidence of the "cost" to be paid by meeting the *wear levelling* requirement. It proceeds by re-writing one side of the target equality ($replay(FS', FI', J')$) into the other (RI'), under the given context:

$$
\begin{array}{ll}
& replay(FS', FI', J') \\
= & \quad \{ \ \text{substitutions enabled by clauses (28) to (36)} \ \} \\
& replay(FS \cup del\ N, FI, J') \\
= & \quad \{ \ \text{definition (15)}\ ;\ \textit{active}\ \text{distributes over union} \ \} \\
& (active\ FS \cup active(del\ N)) \lhd (index\ (FS \cup del\ N) \restriction (\geq_{J'})) \\
= & \quad \{ \ \text{orthogonality (40)}\ ;\ \textit{index}\ \text{distributes over union}\ ;\ \textit{del}\ (41) \ \} \\
& (active\ FS) \lhd ((index\ FS \cup N) \restriction (\geq_{J'})) \\
= & \quad \{ \ \text{split}\ \textit{index}\ FS\ \text{in disjoint parts} \ \} \\
& (active\ FS) \lhd ((index\ FS \cdot \neg \delta N \cup index\ FS \cdot \delta N \cup N) \restriction (\geq_{J'})) \\
= & \quad \{ \ \text{distribution (25) followed by (19)} \ \} \\
& ((active\ FS) \lhd ((index\ FS \cdot \neg \delta N) \restriction (\geq_{J'}))) \cup ((active\ FS) \lhd (((index\ FS \cdot \delta N \cup N) \restriction (\geq_{J'})))) \\
= & \quad \{ \ (21)\ ;\ (42)\ ;\ (34) \ \} \\
& ((active\ FS) \lhd (index\ FS \restriction (\geq_J)) \cdot \Phi_{(\notin S)}) \cup (active\ FS) \lhd ((index\ FS \cdot \delta N \cup N) \restriction (\geq_{J'}))
\end{array}
$$

$$
\begin{array}{ll}
= & \{\ ((index\ FS \cdot \delta N \cup N) \upharpoonright (\geq_{J'}))\ \text{reduces to } N\text{, see (48) below}\ \} \\
& replay(FS, FI, J) \cdot \Phi_{(\notin S)} \cup (active\ FS) \triangleleft N \\
= & \{\ \text{definition (27) ; } \delta(active\ FS) \subseteq \delta FS\ \text{; (32)}\ \} \\
& RI \cdot \Phi_{(\notin S)} \cup \bot \\
= & \{\ \text{clause (30)}\ \} \\
& RI'
\end{array}
$$

□

An important step was left unjustified in the calculation above:

$$(index\ FS \cdot \delta N \cup N) \upharpoonright (\geq_{J'}) = N \upharpoonright J' \tag{48}$$

Prior to its calculation, its intuition: because all addresses in N are greater than those in FS, as granted by M in J', all entries in $index\ FS \cdot \delta N$, which are bound to conflict with N, are overwritten by N.

Let us start from the first remark above and record that, wherever a key participates in both N and FS, its address in N is greater than any other in FS:

$$N \cdot (index\ FS \cdot \delta N)^\circ \subseteq >_{J'} \tag{49}$$

(Proof in appendix A). For improved readability in the calculation of (48) which follows, we abbreviate expressions $index\ FS \cdot \delta N$, $\geq_{J'}$ and $>_{J'}$ by Z, S and S', respectively. Under such abbreviations, (49) shrinks to

$$N \subseteq S'/Z^\circ \tag{50}$$

itself the same as

$$Z \subseteq S'^\circ/N^\circ \tag{51}$$

taking converses. We reason:

$$
\begin{array}{ll}
& (N \cup Z) \upharpoonright S = N \upharpoonright S \\
\Leftrightarrow & \{\ (24)\ \} \\
& (N \upharpoonright S) \cap S/Z^\circ \cup (Z \upharpoonright S) \cap S/N^\circ = N \upharpoonright S \\
\Leftrightarrow & \{\ N \subseteq S/Z^\circ\text{, cf. (50) and } S' \subseteq S\ \} \\
& (N \upharpoonright S) \cup (Z \upharpoonright S) \cap S/N^\circ = N \upharpoonright S \\
\Leftrightarrow & \{\ \text{since } (Z \upharpoonright S) \cap S/N^\circ = \bot\text{, see (52) below}\ \} \\
& N \upharpoonright S = N \upharpoonright S
\end{array}
$$

We are left with

$$(Z \upharpoonright S) \cap S/N^\circ = \bot \tag{52}$$

whose calculation is deferred to appendix A.

6. Last touch in refinement process

A disadvantage of the refined model presented thus far is the growth of journal J, which is bound to cover the whole FS at any time. Power loss recovery of RI by the *replay* function will thus take longer and longer as J grows. This suggests that, from time to time, J should be cleared up while saving the contents of RI

persistently. This is the purpose of FI (flash index), a component of the state model which has played no role in the model so far.

In this way, J will keep only the "difference" between the RI and its *cache* FI, often outdated. Introducing FI requires a *commit* operation which basically updates the contents of FI with the contents of RI and clears J, as specified by the following post-condition:

$J' = \bot$

$FI' = RI$

FS remains unchanged

RI remains unchanged

As a consequence of caching RI, both the concrete invariant *ci* (27) and the *replay* operation (13) need to be upgraded. Earlier on, RI would be rebuilt just by considering J. Now J does not cover the whole FS, only that part changed since the last commit. So, it is necessary to take FI into account by overwriting it with the changes recorded in J. This has the advantage of replaying only the operations that happened after the last check point (*commit*), as captured by re-definition

$$replay(FS, FI, J) = (active\ FS) \lhd (FI \dagger ((J^\circ \lhd (index\ FS)) \upharpoonright (\geq_J))) \tag{53}$$

where $\dagger$ denotes relational overriding [39].

In turn, concrete invariant *ci* calls for further upgrading so as to record extra properties of the state. For instance, it is necessary

- to ensure that FS and FI are consistent by explicitly recapturing (11), for RI replaced by FI;
- to relate J to FI so as to make sure that J only stores changes beyond FI, that is, J only keeps what is "new" with respect to FI.

We omit the formulation and calculation of this extra refinement step from the current paper which, despite the added complexity, would be performed along the same lines and strategy.

A final comment is worthwhile mentioning: operations to be added to the low-level model such as *commit* (sketched above) and *power-loss recovery* (activation of the *replay* function) should have no effect at abstract level [35]. Therefore, extra POs should be discharged ensuring that such operations refine "Skip" [35].

7. Conclusions

Starting by presenting a comprehensive formal methods tool-chain [11] that promotes tool interoperability, this paper focus on its "tuning" with respect to the particular task in hands, aiming at reducing tool integration costs.

Refinement is the particular step in the software development life-cycle which is considered in detail in the paper, resorting to a case study framed in the GC initiative VFS pilot project: the refinement of an abstract file store model into a *journaled* (flash) data model catering for wear leveling and recovery from power loss or unexpected device removal. The exercise shows that such refinement steps can be carried out within a shorter, reduced life-cycle, where model checking in Alloy goes hand in hand with manual proofs carried out in the (pointfree) algebra of binary relations.

We stress that the simplified tool-chain of Figure 3 targets model refinement only. Although the complete tool-chain reported in [11] could be used, the fact that refinement is a systematic process with no subjective user requirements to be validated renders model execution and animation unnecessary. By contrast, generation of counter-examples unveiling defective representation of abstract objects by concrete ones or the violation of low-level constraints is very welcome. In this respect, the particular case study presented in the paper is illustrative of a class of refinement in which the implementation model ends up far more elaborate than first thought due to non-functional requirements (*wear levelling* and *power-loss recovery* in the example). This puts special burden on the preservation of concrete constraints when compared to the actual operations' refinement proofs.

There is a cost to be paid by the proposed simplification, and this is seamless integration of Alloy models with the (pointfree) maths underlying calculations. In this respect, we can report success in following Alloy's lemma "everything is a relation" *avant la lettre*: relational data models in Alloy match well with calculational proofs carried out in the algebra of programming, ie. the relation calculus [1]. By *avant la lettre* we also

mean full respect for Alloy's encoding of data-structures such as lists, for instance. Embarking on the usual inductive proofs about lists would be difficult and counter-productive. Instead, we decided to define new relational combinators capturing declaratively what would, in VDM++ or HOL, for instance, be recursive (non trivial) operations — for instance, that of updating the RAM index by inspecting the list (journal) which keeps all addresses ordered by "creation time". Interestingly enough, we eventually became aware of one such combinator — relation *thinning* ($\upharpoonright$) defined by (20) — being a generalization of a similar operator defined in [1] for algorithmic optimization.

Along the exercise, steps in proofs such as that involving (48), for instance, could be model-checked before their actually being manually calculated, thus providing timely confidence on the overall reasoning and preventing the process from stopping at a particular (difficult) step.

All in all, we have succeeded in proving the refinement of an abstract file store model into its journaled implementation with no need for mechanical theorem proving, just by relying on a very simple tool-chain that supports model-checking and embraces the principles of notation conciseness and correctness by calculation. Judicious use of "Swiss army knife" combinators such as the one just mentioned above was also central to the whole strategy.

8. Related work

Verifiable file system. Since the VFS mini-challenge was put forward, contributions have either focused on verification or refinement, see e.g. [29, 30, 16, 6]. References [29, 30] already contemplate NAND flash memory peculiarities such as wear levelling, erase unit reclamation and tolerance to power loss. More recently, new papers [45, 22, 7, 4, 17] on file system formalization have become available.

Theorem proving is used in [22], which follows a top down approach in formalizing a hierarchical file system. Reference [45] reports on a bottom-up verification of the UBIFS Linux file system for flash memories using the KIV theorem prover. This was the paper that inspired us to model and verify the journaling mechanism using our approach. Event-B and the Rodin tool are used in [7] to formally specify and fully verify a tree-based file system model. The approach is based an a dual notion of refinement: *horizontal* for feature augmentation; and *vertical* for structural refinement. Reference [4] reports on advances in the thorough formalization of a (low level) NAND flash memory based on the Open NAND Flash Interface standards. Not all research related to the verifiable file system project is about designing new file store models. Some researchers have chosen to take existing models one step further, as is the case of [17] in starting from an existing Z specification by Morgan and Sufrin [38]. The outcome is the mechanization of the required proofs using the Z/Eves theorem prover.

Other file system implementations have also been mechanically verified by model checking [18, 52, 30]. Yang *et al* [52] found several errors in widely used file system implementations that were reported back to developers. Galloway and others [18] analyzed a concurrent model of the Linux Virtual File System, which bridges between the Linux kernel and the miscellaneous file system implementations that it supports.

Integration of formal tools. Despite the proliferation of independent languages and tools supporting formal specification and verification, efforts are being made towards integrating such tools in development environments that are more and more agile and sophisticated. Good examples of such integration are Alloy4Eclipse [33], the Rodin tool for Event-B [5] and the already mentioned Overture tools for VDM.

Part of the tool-chain presented in the current paper is already implemented in the Overture project, thanks to our work on the APS workflow [12]. Current efforts go into improving interoperability among Overture internal components, the VDMTools and HOL. This will hopefully produce a cross platform proof system capable of mechanically discharging all VDM-standard POs.

Using Alloy to verify refinement steps. Alloy has been used in conjunction with Event-B [34] and Z [3]. In [34] Alloy is used to validate some invariants for which an automatic proof was not achieved through theorem proving. This Event-B and Alloy pairing approach is identical to our mix of VDM++ with Alloy in the complete tool-chain. Reference [3] reports on how Alloy Analyzer's simulation capabilities can be of help in verifying data refinement in Z. Instead of using the state space search capabilities, as in the current paper, the author of [3] relies on the premise that if there is a retrieve relation between abstract and concrete states, then the refinement is sound. So, in this case, the Alloy Analyzer was required to produce

instances of the retrieve relation to verify the refinement, in contrast to our approach where it is the absence of counter-examples that builds confidence that a correction proof is within reach.

9. Future work

Need for comparative work. As already mentioned, the research presented in this paper was greatly inspired by the work of [45] where a model of the UBIFS is fully verified using the KIV theorem prover. It will be interesting to tally up the two approaches in order to obtain some comparative data. To achieve this, one needs to be sure that both models under comparison are in fact the same. For this purpose, and in parallel with the research presented in this paper, the KIV model of [45] is currently being faithfully translated to Alloy [8]. Once this translation becomes available we intend to check how much of its Alloy can be re-written in the pointfree style, and check the feasibility of carrying out the corresponding pointfree proofs by hand. Should this be unfeasible, one might invest into theorem provers of a different kind: those which, as is the case of Prover9, work directly with algebraic structures of the kind one plays with in relational algebra. Some thoughts on linking Alloy to Prover9 can already be found in [25]. On the other hand, the possibility of [8] eventually granting a direct link from Alloy to KIV is open, paving the way to integrating this theorem prover in our tool-chain.

Model slicing for modular proofs. One of our earliest intuitions about tool-chained verification is the "single-PO, multiple-proof-technology" approach already reported in [10]. In principle this would lead to simpler proofs in the same way program slicing [51] leads to simpler program analyses because the "interference" of all operations put together is factored out. Think for instance of components of the state invariant which are required by one operation but not by the others. When verifying models whose development is out of the verifier control, model *slicing* tools will be of great value, since they can isolate the smallest sub-model that accommodates some target property, operation or data type. This is another aspect which calls for automation: operation-wise manual slicing carried throughout the project [10, 9] has proved to be very time-consuming.

Generic and induction-free proofs. A key aspect of the refinement step calculated in the current paper is its genericity: the intrinsics of types $Path$ and $File$ play no role in calculations, as purported by leaving variables D and K unspecified throughout the exercise. In other words, any other *simple* relation of type $D \xleftarrow{\quad} K$ is covered by the given journaling refinement step. Knowing that any finite, abstract data model (possibly recursive) can be refined into a collection of such simple relations [39], and that any of these can be "journaled" in the way described in the current paper, one will be left with a generic approach to "data model journaling". Of course, provided that refinement proofs are carried out for all CRUD (create, read, update, delete) operations on a simple relation, as already suggested in [39] in a different context.

Another line of research stemming from the given refinement step is the thorough exploitation of the *thinning* combinator (20) which makes a "selection of pairs" from some relation R according to the criteria captured by some other relation S (not just maximization) and leads to induction-less proofs such as those carried out in the paper. Furthermore, thanks to property (25) it is highly parallelizable and thus ideal for high performance data processing. We hypothesize that this operator can be applied in a much wider domain and is sufficiently generic to have a positive impact on the way we reason about data models.

References

[1] R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.

[2] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[3] C. Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electr. Notes Theor. Comput. Sci.*, 137(2):23–44, 2005.

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

[5] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (Rigorous open Development Environment for Complex Systems). In *WORDS*, pages 23–26. IEEE Computer Society, 2005.

[6] K. Damchoom, M. Butler, and J. Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In S. Liu, T.S.E. Maibaum, and K. Araki, editors, *ICFEM*, volume 5256 of *LNCS*, pages 25–44. Springer, 2008.

[7] K. Damchoom and M.J. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In Oliveira and Woodcock [42], pages 134–152.

[8] M.J. Fernandes. Formal Verification using the ESC-PF calculus and model-checking in Alloy of the UBIFS File System for Flash Memory. Master's thesis, University of Minho, Informatics Department, 2010. In preparation.

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

[10] M. Ferreira, S. Silva, and J.N. Oliveira. Verifying Intel Flash File System Core Specification. Technical Report CS-TR-1099, Newcastle University, May 2008.

[11] M. A. Ferreira and J.N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In Oliveira and Woodcock [42], pages 153–169.

[12] M.A. Ferreira. Implementing the Overture Automatic Proof System. Technical Report CS-TR-1177, Newcastle University, November 2009.

[13] J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

[14] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.

[15] J. Fitzgerald, P.G. Larsen, and S. Sahara. VDMTools: advances in support for formal modeling in VDM. *SIGPLAN Notices*, 43(2):3–11, 2008.

[16] L. Freitas, Z. Fu, and J. Woodcock. POSIX file store in Z/Eves: An experiment in the verified software repository. In *ICECCS '07*, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[17] L. Freitas, J. Woodcock, and Zheng Fu. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. Comput. Program.*, 74(4):238–257, 2009.

[18] A. Galloway, G. Lüttgen, J.T. Mühlberg, and R. Siminiceanu. Model-checking the Linux virtual file system. In N.D. Jones and M. Müller-Olm, editors, *VMCAI*, volume 5403 of *LNCS*, pages 74–88. Springer, 2009.

[19] M. Gordon. *From LCF to HOL: a short history*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[20] M.J.C. Gordon. Introduction to the HOL System. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *TPHOLs*, pages 2–3. IEEE Computer Society, 1991.

[21] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Asp. Comput.*, 20(1):41–59, 2008.

[22] W.H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. *Electr. Notes Theor. Comput. Sci.*, 259:67–85, 2009.

[23] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[24] T. Hoare and J. Misra. Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project. In *VSTTE*, pages 1–18, 2005.

[25] P. Höfner and G. Struth. On automating the calculus of relations. In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008.

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

[27] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

[28] R. Joshi and G. J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. In Meyer and Woodcock [36], pages 49–56.

[29] E. Kang and D. Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In E. Börger, M. Butler, J.P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, pages 294–308. Springer, 2008.

[30] E. Kang and D. Jackson. Designing and Analyzing a Flash File System with Alloy. *International Journal of Software and Informatics*, 3(1):129–148, 2009.

[31] P.G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture initiative integrating tools for VDM. *ACM SIGSOFT Soft. Eng. Notes*, 35(1):1–6, 2010.

[32] P.G. Larsen, K. Lausdahl, and N. Batle. Combinatorial Testing for VDM++, 2009. Submitted for publication.

[33] D. Leberre and F. Delorme. An Eclipse plugin for the Alloy4 tool. Website: `http://code.google.com/p/alloy4eclipse/`.

[34] P.J. Matos and J. Marques-Silva. Model Checking Event-B by Encoding into Alloy. *CoRR*, abs/0805.3256, 2008.

[35] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3):197–239, 2009. June/September.

[36] B. Meyer and J. Woodcock, editors. *Verified Software: Theories, Tools, Experiments, First IFIP Conference, VSTTE 2005, Zurich, Switzerland*, volume 4171 of *LNCS*. Springer, 2008.

[37] C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C.A.R. Hoare, series editor.

[38] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. *IEEE Trans. Soft. Eng.*, 10(2):128–142, 1984.

[39] J.N. Oliveira. Transforming Data by Calculation. In *GTTSE'07*, volume 5235 of *LNCS*, pages 134–195. Springer, 2008.

[40] J.N. Oliveira. Extended Static Checking by Calculation using the Pointfree Transform. In A. Bove et al., editor, *LerNet ALFA Summer School 2008*, volume 5520 of *LNCS*, pages 195–251. Springer-Verlag, 2009.

[41] J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'04*, volume 3125 of *LNCS*, pages 334–356. Springer, 2004.

[42] M.V. Oliveira and J. Woodcock, editors. *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, volume 5902 of *Lecture Notes in Computer Science*. Springer, 2009.
[43] Larsen P.G, J.S. Fitzgerald, and S. Riddle. Practice-oriented courses in formal methods using VDM++. *Formal Asp. Comput.*, 21(3):245–257, 2009.
[44] N. Plat and P.G. Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Notices*, 27(8):76–82, 1992.
[45] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract specification of the ubifs file system for flash memory. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 190–206. Springer, 2009.
[46] K. Slind and M. Norrish. A Brief Overview of HOL4. In O.A. Mohamed, C.M., and S. Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
[47] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[48] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Math. Soc., 1987. AMS Colloq. Pub., v. 41, Providence, Rhode Island.
[49] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.
[50] S. Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University, Computer Science Department, 2007.
[51] M. Weiser. Program slicing. In *5th Int. Conf. on Software Eng., San Diego, California*, pages 439–449. IEEE Computer Society Press, March 1981.
[52] J. Yang, P. Twohey, D.R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

A. Appendix

A.1. Basic results of relation algebra

Converses:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{54}$$
$$(R^\circ)^\circ = R \tag{55}$$
$$R \subseteq R \cdot R^\circ \cdot R \tag{56}$$

Simplicity of union:

$$R \cup S \text{ is simple} \quad \Leftrightarrow \quad R \text{ is simple} \wedge S \text{ is simple} \wedge R \cdot S^\circ \subseteq id \tag{57}$$

Injectivity of union:

$$R \cup S \text{ is injective} \quad \Leftrightarrow \quad R \text{ is injective} \wedge S \text{ is injective} \wedge R^\circ \cdot S \subseteq id \tag{58}$$

Shunting rules for relations and functions:

$$R \cdot X \subseteq S \Leftrightarrow R \subseteq S/X \tag{59}$$
$$R \cdot f^\circ \subseteq S \Leftrightarrow R \subseteq S \cdot f \tag{60}$$

In case of M injective:

$$M^\circ \cdot X \subseteq S \Leftrightarrow \rho M \cdot X \subseteq M \cdot S \tag{61}$$
$$X \cdot M \subseteq S \Leftrightarrow X \cdot \rho M \subseteq S \cdot M^\circ \tag{62}$$

Domain and range (for Φ coreflexive),

$$\delta R \subseteq \Phi \Leftrightarrow R \subseteq \top \cdot \Phi \tag{63}$$
$$\rho R \subseteq \Phi \Leftrightarrow R \subseteq \Phi \cdot \top \tag{64}$$

from which a number of properties arise, namely:

$$\rho R = \delta(R^\circ) \tag{65}$$
$$\top \cdot \delta R = \top \cdot R \tag{66}$$
$$\rho R \cdot \top = R \cdot \top \tag{67}$$
$$R \cdot \delta R = R \tag{68}$$
$$R = (\rho R) \cdot R \tag{69}$$
$$\Phi \subseteq \Psi \Leftrightarrow \Phi \subseteq \top \cdot \Psi \tag{70}$$
$$\delta R \subseteq \delta S \Leftrightarrow R \subseteq \top \cdot S \tag{71}$$
$$\delta(R \cdot S) = \delta(\delta R \cdot S) \tag{72}$$
$$\rho(R \cdot S) = \rho(R \cdot \rho S) \tag{73}$$

A.2. Proofs left pending in main text

Calculation of (26):

$$\begin{array}{ll} & \geq_J \cap \geq_J{}^\circ \subseteq id \\ \Leftrightarrow & \quad \{ \text{ (16) twice } \} \\ & (J \cdot \geq \cdot J^\circ) \cap (J \cdot \geq^\circ \cdot J^\circ) \subseteq id \\ \Leftrightarrow & \quad \{ \text{ distribution, since } J \text{ is injective } \} \\ & J \cdot (\geq \cap \geq^\circ) \cdot J^\circ \subseteq id \\ \Leftarrow & \quad \{ \text{ since } J \text{ is simple } \} \\ & J \cdot (\geq \cap \geq^\circ) \cdot J^\circ \subseteq J \cdot J^\circ \\ \Leftarrow & \quad \{ \text{ monotonicity } \} \\ & \geq \cap \geq^\circ \subseteq id \end{array}$$

□

Calculation of (38):

$$\begin{array}{ll} & FS' = FS \cup (del\ N) \text{ is simple} \\ \Leftrightarrow & \quad \{ \text{ definition; simplicity of union (57) } \} \\ & \left\{ \begin{array}{l} FS \text{ is simple} \\ \langle id, iDEL \rangle \cdot N^\circ \text{ is simple} \\ \langle id, iDEL \rangle \cdot N^\circ \cdot FS^\circ \subseteq id \end{array} \right. \\ \Leftrightarrow & \quad \{ \ FS \text{ and } N^\circ \text{ are simple ; splits of functions are functions } \} \\ & \langle id, iDEL \rangle \cdot N^\circ \cdot FS^\circ \subseteq id \\ \Leftrightarrow & \quad \{ \text{ converses (54) ; shunting rule (60) } \} \\ & FS \cdot N \subseteq \langle id, iDEL \rangle \\ \Leftrightarrow & \quad \{ \text{ (32) } \} \\ & \bot \subseteq \langle id, iDEL \rangle \\ \Leftrightarrow & \quad \{ \ \bot \text{ is below anything } \} \\ & true \end{array}$$

□

Calculation of (39):

$$
\begin{array}{ll}
 & J' = J \cup M \text{ is simple} \\
\Leftrightarrow & \quad \{ \text{ (57), since } J \text{ and } M \text{ are simple } \} \\
 & J \cdot M^\circ \subseteq id
\end{array}
$$

This is granted by condition (35) ensuring that M is right-appended to J, since the position of every address in M is strictly larger than that of any address in J. In fact, the stronger condition

$$J \cdot M^\circ \subseteq \bot \tag{74}$$

stems from (35):

$$
\begin{array}{ll}
 & M^\circ \cdot \top \cdot J \subseteq > \\
\Rightarrow & \quad \{ \text{ since } > \text{ is ireflexive } \} \\
 & M^\circ \cdot \top \cdot J \cap id \subseteq \bot \\
\Rightarrow & \quad \{ \text{ monotonicity } \} \\
 & J \cdot (M^\circ \cdot \top \cdot J \cap id) \cdot M^\circ \subseteq \bot \\
\Leftrightarrow & \quad \{ \text{ distribution, since } J \text{ is injective and } M^\circ \text{ is simple } \} \\
 & J \cdot M^\circ \cdot \top \cdot J \cdot M^\circ \cap J \cdot M^\circ \subseteq \bot \\
\Rightarrow & \quad \{ \top \text{ is above everything ; transitivity } \} \\
 & J \cdot M^\circ \cdot (J \cdot M^\circ)^\circ \cdot J \cdot M^\circ \cap J \cdot M^\circ \subseteq \bot \\
\Leftrightarrow & \quad \{ \text{ (56) } \} \\
 & J \cdot M^\circ \subseteq \bot \\
\Box &
\end{array}
$$

Calculation of (41):

$$
\begin{array}{ll}
 & index(del\ N) \\
\Leftrightarrow & \quad \{ \text{ inline definitions } \} \\
 & (\pi_1 \cdot \langle id, iDEL \rangle \cdot N^\circ)^\circ \\
\Leftrightarrow & \quad \{ \text{ cancellation of split by projection } \pi_1 \ \} \\
 & (id \cdot N^\circ)^\circ \\
\Leftrightarrow & \quad \{ \text{ converses (54) } \} \\
 & N \\
\Box &
\end{array}
$$

Calculation of (42):

$$
\begin{array}{ll}
 & index\ FS \upharpoonright (\geq_{J'}) \\
= & \quad \{\ (22)\ \} \\
 & index\ FS \upharpoonright (\rho(index\ FS) \cdot J' \cdot \geq \cdot J'^{\circ} \cdot \rho(index\ FS)) \\
= & \quad \{\ \rho(index\ FS) = \delta FS\ (13)\ \} \\
 & index\ FS \upharpoonright (\delta FS \cdot J' \cdot \geq \cdot J'^{\circ} \cdot \delta FS) \\
= & \quad \{\ (28)\ ;\ (32)\ ;\ (33)\ \} \\
 & index\ FS \upharpoonright (\delta FS \cdot J \cdot \geq \cdot J^{\circ} \cdot \delta FS) \\
= & \quad \{\ (13) \text{ again, followed by } (22)\ \} \\
 & index\ FS \upharpoonright (J \cdot \geq \cdot J^{\circ}) \\
= & \quad \{\ \text{definition}\ \} \\
 & index\ FS \upharpoonright (\geq_{J}) \\
\Box &
\end{array}
$$

Calculation of (45):

$$
\begin{array}{ll}
 & \delta FS' \\
= & \quad \{\ \cup\text{-distribution ; clause } (29)\ ;\ ci\ (27)\ \} \\
 & \rho J \cup \delta(del\ N) \\
= & \quad \{\ (72)\ \} \\
 & \rho J \cup \delta N^{\circ} \\
= & \quad \{\ (65)\ ;\ (33)\ \} \\
 & \rho J \cup \rho M \\
= & \quad \{\ \cup\text{-distribution ; } (28)\ \} \\
 & \rho J' \\
\Box &
\end{array}
$$

Calculation of (46): injectivity of J' (28,58) requires $M^{\circ} \cdot J \subseteq id$, which is granted by (74):

$$
\begin{array}{ll}
 & J \cdot M^{\circ} \subseteq \bot \\
\Leftrightarrow & \quad \{\ \text{monotonicity}\ \} \\
 & M^{\circ} \cdot J \cdot M^{\circ} \cdot M \subseteq \bot \\
\Leftrightarrow & \quad \{\ M \text{ is injective; transitivity}\ \} \\
 & M^{\circ} \cdot J \subseteq \bot \\
\Box &
\end{array}
$$

Calculation of (49):

$$
\begin{array}{ll}
& N \cdot (index\ FS \cdot \delta N)^\circ \subseteq >_{J'} \\
\Leftrightarrow & \quad \{\ (10)\ ;\ \text{converses}\ \} \\
& N \cdot \delta N \cdot \pi_1 \cdot FS \subseteq >_{J'} \\
\Leftrightarrow & \quad \{\ N \cdot \delta N = N\ ;\ (28)\ ;\ (16)\ \} \\
& N \cdot \pi_1 \cdot FS \subseteq (J \cup M) \cdot > \cdot (J \cup M)^\circ \\
\Leftrightarrow & \quad \{\ (22)\ \text{and}\ N,\ FS\ \text{orthogonal to respectively}\ J,\ M\ \} \\
& N \cdot \pi_1 \cdot FS \subseteq M \cdot > \cdot J^\circ \\
\Leftrightarrow & \quad \{\ (69),\ (33)\ \text{and}\ (65)\ ;\ (12)\ \text{and}\ (68)\ \} \\
& \rho M \cdot N \cdot \pi_1 \cdot FS \cdot \delta J \subseteq M \cdot > \cdot J^\circ \\
\Leftrightarrow & \quad \{\ \text{shunting}\ (61{,}62)\ \} \\
& M^\circ \cdot N \cdot \pi_1 \cdot FS \cdot J \subseteq > \\
\Leftarrow & \quad \{\ N \cdot \pi_1 \cdot FS \subseteq \top\ ;\ \text{transitivity}\ \} \\
& M^\circ \cdot \top \cdot J \subseteq > \\
\Leftrightarrow & \quad \{\ (35)\ \} \\
& true
\end{array}
$$

□

Calculation of (52):

$$
\begin{array}{ll}
& (Z \upharpoonright S) \cap S/N^\circ = \bot \\
\Leftrightarrow & \quad \{\ (51)\ \} \\
& (Z \upharpoonright S) \cap S/N^\circ \cap S'^\circ/N^\circ = \bot \\
\Leftrightarrow & \quad \{\ \text{division preserves}\ \cap\ \} \\
& (Z \upharpoonright S) \cap (S \cap S'^\circ)/N^\circ = \bot \\
\Leftrightarrow & \quad \{\ S \cap S'^\circ = \bot\ \} \\
& (Z \upharpoonright S) \cap \bot/N^\circ = \bot \\
\Leftrightarrow & \quad \{\ \text{indirect equality, for all suitably typed}\ X\ \} \\
& X \subseteq (Z \upharpoonright S) \wedge X \subseteq \bot/N^\circ\ \Leftrightarrow\ X \subseteq \bot \\
\Leftrightarrow & \quad \{\ \text{expanding definitions}\ \} \\
& X \subseteq Z \wedge X \cdot Z^\circ \subseteq S \wedge X \cdot N^\circ \subseteq \bot\ \Leftrightarrow\ X \subseteq \bot \\
\Leftrightarrow & \quad \{\ \text{since}\ Z\ \text{and}\ N\ \text{are not domain-disjoint}\ \} \\
& X \subseteq \bot\ \Leftrightarrow\ X \subseteq \bot
\end{array}
$$

□