Alloy Meets the Algebra of Programming: A Case Study

José N. Oliveira and Miguel A. Ferreira, *Member*, *IEEE*

Abstract—Relational algebra offers to software engineering the same degree of conciseness and calculational power as linear algebra in other engineering disciplines. Binary relations play the role of matrices with similar emphasis on multiplication and transposition. This matches with Alloy's lemma "everything is a relation" and with the relational basis of the Algebra of Programming (AoP). Altogether, it provides a simple and coherent approach to checking and calculating programs from abstract models. In this paper, we put Alloy and the Algebra of Programming together in a case study originating from the Verifiable File System mini-challenge put forward by Joshi and Holzmann: verifying the refinement of an abstract file store model into a *journaled* (FLASH) data model catering to wear leveling and recovery from power loss. Our approach relies on diagrams to graphically express typed assertions. It interweaves model checking (in Alloy) with calculational proofs in a way which offers the best of both worlds. This provides ample evidence of the positive impact in software verification of Alloy's focus on relations, complemented by induction-free proofs about data structures such as stores and lists.

Index Terms—Model checking, algebra of programming, software verification, grand challenges in computing

1 INTRODUCTION

This paper is concerned with a major topic in software engineering: that of designing correct programs in the first place. Let us begin by inquiring into the phrase *software engineering* itself. The terminology seems to date from the Garmisch NATO conference in 1968, from whose report [1] the following excerpt is quoted:

> *In late 1967, the Study Group recommended the holding of a working conference on Software Engineering. The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.*

Provocative or not, the need for sound theoretical foundations has clearly been of concern since the very beginning of the discipline. However, how "scientific" do such foundations turn out to be, now that four decades have since elapsed?

Many took the provocation seriously and embarked on researching *formal methods* for developing code from formal specifications. However, in a recent paper [2], David Parnas questions such methods, which he regards as still unfit for the software industry:

> *We must learn to use mathematics in software development, but we need to question, and be prepared to discard, most of the methods that we have been discussing and promoting for all these years.*

- J.N. Oliveira is with the High-Assurance Software Laboratory (HASLab)/INESC TEC, Departamento de Informática, University of Minho, Campus de Gualtar, Braga 4710-057, Portugal. E-mail: jno@di.uminho.pt.
- M.A. Ferreira is with the Software Improvement Group, Rembrandt Tower, 15th Floor, Amstelplein 1, 1096 HA Amsterdam, The Netherlands. E-mail: m.ferreira@sig.eu.

Manuscript received 23 Feb. 2011; revised 29 Dec. 2011; accepted 21 Jan. 2012; published online 17 Feb. 2012.
Recommended for acceptance by M. Chechik.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-02-0055.
Digital Object Identifier no. 10.1109/TSE.2012.15.

At the core of Parnas objections lies the contrast between the current ad hoc (re)invention of burdening mathematical notation and elegant concepts which are neglected, often for cultural reasons or (lack of) background.

The question is: What is it that tells "good" and "bad" methods apart? As Parnas writes, *there is a disturbing gap between software development and traditional engineering disciplines*. In such disciplines one finds a successful, well-established mathematical background essentially made of calculus, linear algebra, and probability theory.

Central to engineering mathematics is the construction of sets of simultaneous equations as models of physical systems (e.g., circuits, etc.):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{1m}x_m & = & b_1, \\ \vdots & \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{nm}x_m & = & b_n, \end{cases} \tag{1}$$

that is, formulæ of the form

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} a_{ij}x_j = b_i. \tag{2}$$

The maturity of traditional engineering mathematics can be appreciated from the fact that such (often very large) sets of equations do not intimidate engineers, thanks to the *matrix* and *vector* concepts: Grouping all coefficients a_{ij} of (1) in a matrix A, variables x_j in a vector X, and values b_i in a vector B, (1) becomes

$$A \cdot X = B,$$

where operator $(\cdot)$ denotes matrix multiplication. Backhouse [3] writes: *"In this way a set of equations has been reduced to a single equation. This is a tremendous improvement in concision that does not incur any loss of precision!"*

Thus, notation scales up and *quantity* does not disturb *quality*. Another sign of maturity arises from the use of

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

mathematical transformations, such as the Laplace transform [4], which changes the "mathematical space" so as to convert "difficult" sets of equations (e.g., differential) into "easy" ones (e.g., polynomial), whose solutions are mapped back to the original problem domain by the converse transform. Once again, complexity is controlled via effective mathematical techniques.

One may wonder about parallels to these techniques in formal methods for software design in their use of formal logics. Do such logics scale up to very large sets of clauses issued by proof obligation (PO) generators, for instance? Is there a *linear algebra* for logic and set theory? Is there a logic equivalent to a *matrix*?

While the answer to the first question is *poorly!*, those to the other questions are affirmative: *yes, there are!* Quoting [2] once again:

> *There is an alternative. Some researchers have been studying the use of relational methods in computer science; (...) the well-known laws of relational algebra can serve as the axiomatic basis for programming. The axioms of relational algebra are simple and universal. This approach seems to have been neglected by most mainline researchers in the area of formal methods.*

Binary relations are Boolean matrices, thus providing a straight parallel with linear algebra. And the relational composition of two relations R and S, usually denoted by the same multiplicative term $R \cdot S$, provides another one. In set theory, this relational operator is defined indirectly as follows, assuming the *set-of-pairs* understanding of binary relations: Pair (b, c) is in $R \cdot S$ iff there exist one or more mediating a such that $(b, a) \in R$ and $(a, c) \in S$.

If we look at one of the first definitions of this combinator, due to Charles Peirce (1839-1914) and explained in [5], we realize that it computes inner products like those of (2), where multiplication (restricted to 0s and 1s) captures logical conjunction and addition (respectively, summation) captures disjunction (respectively, existential quantification), if clipped at 1. Thereafter, relation union $R \cup S$ is nothing but index-wise Boolean matrix addition and distributive laws

$$R \cdot (S \cup T) = (R \cdot S) \cup (R \cdot T), \tag{3}$$

$$(S \cup T) \cdot R = (S \cdot R) \cup (T \cdot R) \tag{4}$$

arise from the *bilinearity* of the underlying matrix algebra.

In his recent book on *relational mathematics* [6], Gunther Schmidt makes extensive use of matrix notation, concepts and operations in relation algebra. An account of the work on calculational reasoning about regular and Kleene algebras of matrices can be found in [3]. The close relationship between categories of matrices and relations is implicit in the allegorial setting of Freyd and Ščedrov [7]: Essentially, matrices whose data values are taken from *locales* (e.g., the Boolean algebra of truth values) are the morphisms of the corresponding allegory (e.g., that of binary relations).

The application of this allegorial view of relational algebra to the synthesis of algorithms has led to a new discipline, called *Algebra of Programming* (AoP), which has reached textbook format in [8]. This textbook provides ample evidence of the usefulness of relation-algebra-based reasoning in algorithm design.

Influenced by this effectiveness, we (and others) have reported success in its application to other areas of the software sciences, namely, software components [9], coalgebraic reasoning [10], algorithmic refinement [11], data model refinement and relational database design [12], separation logic [13], extended static checking (ESC) [14], and data dependency theory [15].

The main purpose of this paper is to show the application of relational calculation to software verification. A case study is presented which can be framed in the challenge put forward by Joshi and Holzmann [16] on developing a reliable FLASH file-system for in-flight software. The challenge has been widely tackled in the literature (see Sections 6 and 10). We ourselves have addressed it before, from two rather different angles: a tool-intensive approach based on a verification life-cycle supported by a tool-chain combining several off-the-shelf technologies [17] and a minimalist one [14] in which only one such tool—Alloy [18]—is kept, blending nicely with AoP calculation.

The current paper follows the latter approach and illustrates a simple, but effective methodology for software design from abstract models which is currently adopted and taught at Minho University, at the postgraduate level. It combines the pragmatism of Alloy's often-called *lightweight approach* to formal methods [18] with AoP calculation [8], providing formal proofs of those desirable properties of the design which "survive" the model-checking phase, using the Alloy tool.

In this way, the best of both worlds is met: model-checking filters wrong intuitions and careless errors, upon which AoP gets into the play and completes the certification of the design, "by calculation."

Paper structure. We start by briefly introducing *binary relations* as a device for thinking about software (Section 2), leading to an introduction to the AoP (Section 3), and to Alloy (Section 4). Preparing for the paper's case study, a brief account of verification theory is given in Section 5.

Section 6 introduces the VFS mini-challenge and Section 7 focuses on the paper's case study: a relational model of a journaled FLASH store. The link from the abstract level to the refined (journaled) level is presented in Section 8, including data type invariants and the correctness of the operation that deletes items from the file store. Section 9 gives the last touch in the journaling mechanism by introducing in-device caching to increase fault-recovery performance. The reader less interested in details of the refinement case study may want to skip Sections 8 and 9 on first reading.

Sections 10 and 12 give a review of related and future work, respectively. An account of advantages and limitations of the proposed strategy is given in Section 11. Appendix A.1 lists auxiliary Alloy functions. Standard facts of relation algebra and proofs of side-stream results are deferred to in Appendices A.2 and A.3, respectively.

The target audience is assumed to be aware of abstract modeling and verification techniques.

2 Thinking Relationally

2.1 Relational Ubiquity

It is hard to find a text where the word "relation" does not turn up, and even harder to find one in which no particular

relationship can be found implicit in its semantics, recorded in natural language. Sentences such as *"John is the father of Mary," "Mary speaks English,"* for instance, express *typed* relationships among objects: *fatherhood* between people, *tongue* between people and languages, etc.

Relational sentences like the above have been absorbed by mathematical notation a long time ago, for instance in writing $1 \geq 0$ as an abbreviation of "*1 is at least 0*" and $3 = 1 + 2$ instead of "*3 is the successor of 2*." These examples show that *infix* notation is *the natural* way to write relational facts. In general, notation $b\ R\ a$ (where b and a are objects and R is the relationship) is far better than the set-theory-biased $(b, a) \in R$. And this carries over to the *passive voice*, captured by the *converse* of R, to be denoted R°, which is such that $a\ R^\circ b$ means exactly the same as $b\ R\ a$. For instance, $speaks^\circ = is\ spoken\ by$. In the parallel with linear algebra, converse corresponds to matrix transposition.

The close binding of relational notation to language structure can be further observed in the use of *definite* article "the" in $R = (is\ the\ father\ of)$. This means that, if known, one's father is unique. Relations with this property are referred to as *simple* and satisfy property

$$R \cdot R^\circ \subseteq id, \tag{5}$$

where $(\cdot)$ denotes relation composition (already introduced above), id is the identity relation (that is, $y\ id\ x$ means $y = x$), and $\subseteq$ denotes relational inclusion:

$$R \subseteq S \equiv \forall\ b, a : b\ R\ a \Rightarrow b\ S\ a. \tag{6}$$

2.2 Functions

Definite articles are also implicit in the way one understands facts such as $3 = 1 + 2$ above, where the equality sign expresses the determinism of relation *successor of*. To stress this, one may even define $succ\ n \stackrel{\triangle}{=} 1 + n$ and write $3 = succ\ 2$. Note that the equality sign only makes sense because relation $succ$ is *simple* (deterministic) and *defined* for all its arguments. The property of a relation R being defined for all its inputs is captured by

$$id \subseteq R^\circ \cdot R, \tag{7}$$

in which case we say that R is *entire*.[1]

Functions are relations which are *simple* and *entire* at the same time. Following common practice, functions will be denoted by lower case letters (e.g., f, g) or identifiers starting with such letters (e.g., $succ$). Summing up, mind that infix notation $b f a$ always means $b = f a$, in the case of functions.

2.3 Relation Types

Consistent with relational infix notation, bRa is the declaration of *relation types* in AoP using arrows: $B \stackrel{R}{\longleftarrow} A$ declares binary relation R with source type A and target type B. (For instance, $B = People$ and $A = Language$ in the case of $R = speaks$ above.) We will say that $B \longleftarrow A$ is the *type* of R. Thus, in writing $b\ R\ a$, b (respectively, a) inhabits the target (respectively, source) type of R. Type declarations $B \stackrel{R}{\longleftarrow} A$ and $A \stackrel{R}{\longrightarrow} B$ mean the same thing.

1. Word *total* is avoided so as not to clash with preexisting relational terminology, e.g., *total order*. We adopt this terminology from [7] and [8].

Comparing relations (6) only make sense for relations of the same type. In fact, each relational type $B \longleftarrow A$ forms a Boolean algebra ordered by $\subseteq$ whose least relation is denoted by $\bot$ (empty relation) and largest relation is denoted by $\top$ (topmost relation). Clearly, $b \bot a$ never holds and $b \top a$ is always true.[2]

2.4 Diagrams

Arrow notation makes it possible to express relational formulæ using diagrams. This is a major ingredient of the method because it provides a graphical way of picturing relation types and relational constraints.

Paths in diagrams are built by arrow chaining, which corresponds to relational composition ("... is R of some S of ..." in natural language):

$$B \stackrel{R}{\longleftarrow} A \stackrel{S}{\longleftarrow} C \quad (R \cdot S : B \longleftarrow C) \qquad b(R \cdot S)c\ \equiv\ \exists\ a :\ b\ R\ a\ \wedge a\ S\ c \tag{8}$$

Diagrams arise from comparing paths, for instance,

$$\begin{array}{ccc} Descriptor & \stackrel{FT}{\longleftarrow} & Handle \\ {\scriptstyle path}\downarrow & \subseteq & \downarrow{\scriptstyle \top} \\ Path & \stackrel{FS^\circ}{\longleftarrow} & File \end{array}$$

which depicts constraint

$$path \cdot FT \subseteq FS^\circ \cdot \top, \tag{9}$$

where—already in the file-system modeling domain [14]—simple relation FS models a *file store* (a table that maps file system paths to the respective files), simple FT is the *open-file descriptor* table (a table that holds the information about open files[3]), function $path$ yields the path of a file descriptor, and $\top$ is the largest possible relation between file-handles and files.

2.5 From Diagrams to Logic

What does (9) mean, in predicate logic? We reason

$$\begin{array}{cl} & path \cdot FT \subseteq FS^\circ \cdot \top \\ \equiv & \{\ \text{'at most' ordering (6)}\ \} \\ & \forall\ p,\ h : p(path \cdot FT)h \Rightarrow p(FS^\circ \cdot \top)h \\ \equiv & \{\ \text{composition (8); path is a function}\ \} \\ & \forall\ p, h : (\exists\ d : p = path\ d \wedge d\ FT\ h) \Rightarrow p(FS^\circ \cdot \top)h \\ \equiv & \{\ \text{quantifier calculus} - \text{splitting rule [19]}\ \} \\ & \forall\ d, h : d\ FTh \Rightarrow (\forall\ p : p = path\ d \Rightarrow p(FS^\circ \cdot \top)h) \\ \equiv & \{\ \text{quantifier calculus} - \text{one-point rule [19]}\ \} \\ & \forall\ d, h : d\ FT\ h \Rightarrow (path\ d)(FS^\circ \cdot \top)h. \end{array}$$

2. $\top$ is De Morgan's *"is coexistent with"* relation [5].

3. Open files are manipulated by the file system via open file descriptor data structures, which hold various relevant metadata (e.g., current position within the file). Such descriptors are identified by file handles which the file system provides to applications that manipulate files. This indirection layer avoids unnecessary coupling between applications and the details of the file system implementation.

We still have to unfold term $(path\ d)(FS^\circ \cdot \top)h$:

$$\begin{aligned} & (path\ d)(FS^\circ \cdot \top)h \\ \equiv & \ \{ \text{ composition (8) } \} \\ & \exists\, x : (path\ d) FS^\circ x \wedge x \top h \\ \equiv & \ \{ \text{ converse; } x \top h \text{ always holds } \} \\ & \exists\, x : x\ FS\ (path\ d). \end{aligned}$$

In summary, $path \cdot FT \subseteq FS^\circ \cdot \top$ unfolds into

$$\forall d, h : d\ FT\ h \Rightarrow (\exists\, x : x FS\ (path\ d)). \tag{10}$$

Literally:

If h is the handle of some open-file descriptor d, then this holds the path of some existing file x.

In fewer words:

*Nonexisting files cannot be opened (*referential integrity*).*

Thus, we see how relation diagrams "hide" logically quantified formulæ capturing properties of designs. Another example, in the same domain, is

$$\begin{array}{ccc} Path & \xleftarrow{\ \geq\ } & Path \\ {\scriptstyle FS}\downarrow & \subseteq & \downarrow{\scriptstyle FS} \\ File & \xleftarrow[\ \top\ ]{} & File \end{array} \qquad \text{that is} \qquad FS \cdot \geq \subseteq \top \cdot FS$$

where $\leq$ intends to capture the *child-of* relationship among paths.[4] Similar reasoning will yield a logical interpretation which, in words, will mean that *mother-directories always exist*, another essential property of a file-system.

Easy to draw and memorize are the following (generic) constraints involving two arbitrary relations M and N.

- M, N are domain-disjoint: $M \cdot N^\circ \subseteq \bot$.
- M, N are domain-coherent: $M \cdot N^\circ \subseteq id$.
- The domain of M is strictly above that of N (assuming a strict ordering $>$ on the input types of M and N): $M^\circ \cdot \top \cdot N \subseteq >$.

In summary, relation composition and converse are enough (compare with matrix multiplication and transpose in linear algebra) to capture interesting properties of abstract models, which can be drawn as diagrams. The approach consists of, in the first place, sketching abstract models by drawing diagrams which capture "the things which matter" to a given problem: object types (nodes in the diagrams, usually nouns in the requirements), relationships among such objects (arrows, usually noun phrases in the requirements), and constraints—commutative polygons (e.g., squares, triangles) in diagrams, often corresponding to relative clauses in the requirements.

Clearly, these diagrams only capture static semantics and their role is comparable to that of class diagrams in UML, for instance. But they are far simpler graphically and conceptually, offering a very simple framework for starting a design anew.[5]

4. That is, $p \leq p'$ means that p *is a subpath of* p'.

5. For the comparison to make sense we still need to introduce *products* $(A \times B)$ and coproducts $(A + B)$ of any two given types A, B [8], expressing two dual, standard forms of data aggregation. See Section 4.

3 Algebra of Programming

3.1 Background

Chronologically, relational notation emerged earlier than predicate logic itself in the work of Augustus De Morgan (1806-1871) on binary relations [5]. Later, Peirce (1839-1914) invented quantifier notation to explain De Morgan's algebra of relations (see, e.g., [5] for details). De Morgan's pioneering work was ill fated: The language invented to explain his calculus of relations eventually became more popular than the calculus itself. Alfred Tarski (1901-1983), who had a life-long struggle with quantified notation [20], [21], revived relation algebra. Together with Steve Givant he wrote a book (published posthumously) on *set theory without variables* [22].

Meanwhile, category theory was born, stressing the role of *arrows* and *diagrams* and on the arrow language of diagrams, which is inherently *pointfree*. The category of sets and functions immediately provided a basis for pointfree functional reasoning, but this was by and large ignored by John Backus (1924-2007) in his FP algebra of programs [23]. Anyway, Backus' landmark FP paper was among the first to show how relevant such reasoning style is to computing.

A bridge between the two pointfree schools, the relational and the categorial, was eventually established by Freyd and Ščedrov [7] in their proposal of the concept of an *allegory*. This gave birth to *typed* relation algebra and relation diagrams like those adopted in the current paper. The pointfree algebra of programming as it is understood today [8] stems directly from [7].

3.2 Recent Developments

In the 1990s, the Groningen-Eindhoven MPC group led by Backhouse [3] contributed decisively to the AoP by structuring relation algebra in terms of Galois connections. However intimidating this may sound, it is in fact a great simplification in its structuring the calculus in terms of *rules* which make relational reasoning closer to school algebra.

Think, for instance, of the rule used to reason about whole division of two natural numbers:

$$z \times y \leq x \equiv z \leq x \div y \qquad (y > 0), \tag{11}$$

assumed universally quantified in all its variables. Pragmatically, it expresses a "shunting" rule which enables one to exchange between a whole division in the upper side of an inequality and a multiplication in the lower side. Many properties of $(\times)$ and $(\div)$ can be inferred from (11), for instance, $(x \div y) \times y \leq x$—just replace z by $x \div y$ and simplify.[6]

The parallel with relation algebra is easy to perceive by writing a rule similar to (11):

$$R \cdot S \subseteq X \equiv R \subseteq X/S, \tag{12}$$

which connects *relation division* (an operator which will play a role in the sequel) to relational composition. It can be shown that, while composition hides an existential quantifier (8), division hides a universal one [3]:

6. Rule (11) *connects* division to multiplication, the latter helping to reason about the former. Functions connected in this way are said to be *adjoints*: Multiplication is adjoint of division. Equivalences of this are scalable, powerful devices known in mathematics as Galois connections.

$$c(X/S)a \equiv \forall\, b : a\ S\ b \Rightarrow c\ X\ b. \qquad (13)$$

Thus, (12) captures, in a rather eloquent way, the duality between universal and existential quantification. Moreover, the relational equivalent to $(x \div y) \times y \leq x$ above is $(X/S) \cdot S \subseteq X$. This *cancellation* rule, very often used in practice, unfolds to

$$(\forall\, b : a\ S\ b \Rightarrow\ c\ X\ b) \wedge\ a\ S\ b' \Rightarrow c\ X\ b',$$

that is, to the well-known device in logic known as *modus ponens*: $((S \to X) \wedge S) \to X$.

Often referred to as "shunting rules" [8], equivalences such as (11), (12) are examples of Galois connections. Many other relational concepts are captured in the same way. For instance, the fact that a relation is a function f is equivalent to connection (83) in Appendix A.2. Furthermore, the meaning of two relational operators central to understanding the Alloy semantic rules of Section 4—the domain (δ) and range (ρ) of a relation—are captured by connections (86) and (87) in the same appendix, respectively.

There is something else concerning the two operators just above: They yield sets represented by fragments of the identity relation. In general, for each predicate p, we define binary relation Φ_p 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, $\Phi_p \subseteq id$. This is why these relations are termed *partial identities* or *coreflexives* [8]. Given a set S, the coreflexive which represents S is $\Phi_{\in S}$, also denoted by Φ_S. This is referred to in [18] as the *identity on set* S.

3.3 More on Terminology

We have seen that relations can be simple (5), entire (7), and both (functions). Taking converses, we obtain the dual notions of injectivity

$$R^\circ \cdot R \subseteq id, \qquad (14)$$

and surjectivity

$$id \subseteq R \cdot R^\circ. \qquad (15)$$

Mind the following four AoP rules of thumb easy to infer from the definitions [14].

- Converse of *injective* is *simple* (and vice versa).
- Converse of *entire* is *surjective* (and vice versa).
- Smaller than injective (simple) is injective (simple).
- Larger than entire (surjective) is entire (surjective).

3.4 Remark on Notation

The following conventions will be adopted for saving parentheses in relational expressions, concerning infix operators (such as, e.g., composition, $\cup$) and unary ones (e.g., converse, domain, and range): 1) Unary and prefix operators bind tighter than binary, 2) "multiplicative" binary operators (e.g., composition, $\cap$, $/$) bind tighter than "additive" ones (e.g., $\cup$), 3) relation composition binds tighter than any other multiplicative operator.

4 Alloy

Alloy is a lightweight modeling language for software design inspired by the Z notation and developed by the Software Design Group at MIT [18]. 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 thinking, as will be illustrated in the sequel.

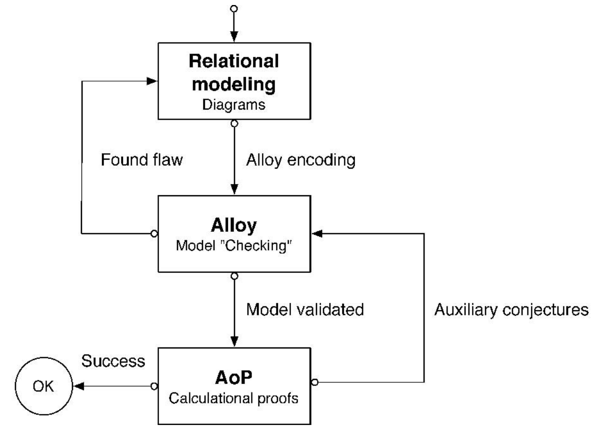


Fig. 1. "Alloy-meets-AoP" verification round-trip.

4.1 Tool Support

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 searching for counter-examples to given assertions.

The Alloy Analyzer helps a great deal in detecting naive, subtle, or special case mistakes. Mistakes of this kind often go undetected by test cases, and are known to jam theorem provers for unforeseeable reasons. The tool relies on the Kodkod model finder to encode Alloy's logic in Boolean logic, which in turn can be subject to fully automated analysis by SAT solvers [18]. This way, Alloy's relational models are mapped onto Boolean formulas that can be either checked for consistency (possible to instantiate) or falsified (existence of 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.

4.2 "Alloy-Meets-AoP" Verification Life-Cycle

The main idea behind blending Alloy with AoP is that of model-checking every proof obligation in a design before proceeding to its correctness proof.

Being a model checker, the Alloy Analyzer does not discharge proofs as such, but is very useful in finding design flaws. The absence of counterexamples builds confidence that a correctness proof is within reach. In this way, one avoids attempting full-fledged proofs of assertions that could be demonstrated impossible by counter-examples. This positive impact on productivity will be addressed later on within our case study. Fig. 1 depicts the corresponding verification life-cycle.

4.3 Alloy-to-AoP Mapping

The main feature of Alloy's notation is the "dot join" combinator, denoted ".", which extends binary relation composition. The formal, relation algebra semantics of the Alloy language has been the subject of some research; see,

e.g., [24], [25], mainly because of the flexibility of "dot join" and the fact that Alloy's relations are n-ary, in general.

Our Alloy-to-AoP correspondence is greatly simplified because relational diagrams restrict to binary relations only, requiring not much more than such relations, converses, "dot join," and set-like operators such as union and intersection. This makes the semantic mapping of our Alloy subset to AoP relatively easy to establish.

4.4 Semantic Rules

Below (part of) Alloy's syntax and respective semantics [18] is explained. We start with the semantic rules which capture its syntax for the *at most* ordering, intersection, union, and converse:

$$[\![R \text{ in } S]\!] = [\![R]\!] \subseteq [\![S]\!], \tag{16}$$

$$[\![R\ S]\!] = [\![R]\!] \cap [\![S]\!], \tag{17}$$

$$[\![R + S]\!] = [\![R]\!] \cup [\![S]\!], \tag{18}$$

$$[\![\tilde{\ }R]\!] = [\![R]\!]^{\circ}. \tag{19}$$

Alloy's syntax for $\top$ is quite interesting in its making the types explicit:

$$[\![A \rightarrow B]\!] = [\![B]\!] \cdot \top \cdot [\![A]\!]. \tag{20}$$

Types A and B are sets which, in our semantics, will be captured by *coreflexives*, a special case of binary relations already introduced in Section 3. In general, given a set $s : A$, we have the semantic rule

$$[\![s]\!] = A \xleftarrow{\Phi_s} A. \tag{21}$$

The largest such s is A itself, represented by the largest such coreflexive: the identity id_A. Writing $s <: iden$ is the standard way in Alloy to obtain coreflexive Φ_s, using the *domain restriction* combinator, defined by semantic rule

$$[\![s <: r]\!] = [\![r]\!] \cdot \Phi_s. \tag{22}$$

Restricted to binary relations, Alloy's "dot join" is (forward) binary relation composition, therefore changing the order in which the producer and consumer relations appear in terms:

$$[\![S.R]\!] = [\![R]\!] \cdot [\![S]\!] \quad C \xleftarrow{[\![R]\!]} B \xleftarrow{[\![S]\!]} A. \tag{23}$$

With these rules we are already in a position to express referential integrity constraint (9) in Alloy syntax:

FT.path **in** (Handle→ File)~FS

Dot join can be used in Alloy between relations which are not binary, e.g., sets (unary relations or vectors) and even scalars (singleton vectors). Recalling the range combinator ρ from Section 3, we have the following semantic rule in the first case:

$$[\![s.R]\!] = \rho([\![R]\!] \cdot [\![s]\!]) \quad C \xleftarrow{[\![s.R]\!]} C \xleftarrow{[\![R]\!]} B \xleftarrow{[\![S]\!]} B, \tag{24}$$

for R binary and s a set (unary relation).[7]

7. Thus, $[\![s]\!]$ is a coreflexive binary relation; recall (21). By resorting to coreflexives we are saved from the *point-density* requirement of [24]. *Points*, as defined in [24], correspond to singleton coreflexives.

Thanks to $[\![R.s]\!] = [\![s.\tilde{\ }R]\!]$ [18] one has $[\![R.s]\!] = \delta([\![s]\!] \cdot [\![R]\!])$, where δ is the domain combinator of Section 3. (Mind the fact that δ and ρ commute through converse, cf. (85) in the Appendix.)

In case R is a function f and s is a scalar x (that is, a singleton vector), $[\![x.f]\!]$ boils down to function application $f(x)$. Thus, the following piece of pointwise Alloy,

all h: Handle, d: h.FT | **some** (d.path).FS,

means the same as the file-system referential integrity constraint given earlier in pointwise notation (10). The Alloy keywords **all** and **some** express the universal and existential quantifiers, respectively. The pipe (|) symbol delimits the declaration of the universally quantified variables. In the above example, the existential quantifier does not have a list of associated variables because it is applied directly to a relation. Some of Alloy's quantifiers, such as the existential, can be used to express multiplicity factors restricting the cardinality of relations. More examples of such multiplicity factors will be provided in the sequel.

One may wonder about which properties of relation composition are preserved by Alloy's "dot join" combinator. Our semantics can be used for this purpose. For instance, let us check whether associativity $(X.R).S = X.(R.S)$ holds in Alloy. In case of binary relations we are done (23). The case of X being unary (a set x) follows:

$$\begin{aligned}
& [\![(x.R).S]\!] \\
& \quad = \{\text{rule (24) twice}\} \\
& \qquad \rho([\![S]\!] \cdot \rho([\![R]\!] \cdot [\![x]\!])) \\
& \quad = \{\text{range of composition (94)}\} \\
& \qquad \rho([\![S]\!] \cdot ([\![R]\!] \cdot [\![x]\!])) \\
& \quad = \{\text{composition is associative (75)}\} \\
& \qquad \rho(([\![S]\!] \cdot [\![R]\!]) \cdot [\![x]\!]) \\
& \quad = \{R \text{ and } S \text{ are binary (23); (24)}\} \\
& \qquad [\![x.(R.S)]\!].
\end{aligned}$$

In a similar way, the interested reader may check that once the middle component is unary, associativity $(R.x).S = R.(x.S)$ requires side condition $\tilde{\ }R.S = S.\tilde{\ }R$.[8]

4.5 Typed Relations in Alloy

Thus far we have seen how close Alloy syntax is to relational AoP notation, making the translation from one to the other almost direct. Alloy's lemma "everything is a relation" carries the analogy further, leading into the way Alloy caters for relation types, using keyword **sig** (for *signature*):

sig A {R : B}

This declares relation R of type $A \longrightarrow B$.

Multiplicity constraints can be added to this syntax in order to capture relation subclasses. For instance,

sig A {R : **lone** B}

declares a *simple* relation

$$A \xrightarrow{R} B, \tag{25}$$

8. This softens the side condition given in [24] for "dot join" associativity to take place.

by addition of multiplicity constraint **lone** (read: *less or equal to one*) to the target type. Note the use of "harpoon" arrows ($\rightharpoonup$) to depict these relations, already adopted in [14] and [17]. This is a way of singling out this important kind of relation in diagrams.

Another such keyword—**one** (read: *exactly one*)—will ensure R *simple* and *entire*, that is, a function:

sig A {f : **one** B}

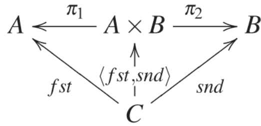
More than one typed relation can be declared in a single Alloy signature, for instance,

sig C {fst : **one** A, snd : **one** B}

or

sig System {
store: Path → **lone** File,
table: Handle → **lone** Descriptor
}

In the first case, one is declaring a pair of functions sharing the same source type C. This is captured by the diagram below, which relates C to the Cartesian product type $A \times B$, where projections π_1 and π_2 are such that $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$, and AoP combinator $\langle f, g\rangle$ pairs the results of its argument functions f and g.[9]



Something else is going on in the second Alloy sample above: Two simple relations *store* and *table* are declared, but their domains are Cartesian products

$$System \times Path \xrightarrow{store} File,$$
$$System \times Handle \xrightarrow{table} Descriptor.$$

Alloy implicitly relies on the "currying" isomorphism,[10]

$$(A \times B) \rightharpoonup C \cong (B \rightharpoonup C)^A,$$

in handling relations of the left-hand side type as *functions* of the right-hand side type, which are higher order in the sense of yielding relations (of type $B \rightharpoonup C$) as results. Thus, for each s of type $System$, $s.store$ is a relation of type $Path \rightarrow File$. Similarly, $Handle \xrightarrow{s.table} Descriptor$ is well typed.

As an instance of this notation, check the following Alloy predicate:

pred ri[s: System] {s.table.path **in** (Handle→ File) ~(s.store)}

which rephrases the referential integrity constraint (9) in terms of signature $System$.[11]

Summing up, note how this semantic interpretation of Alloy's syntax once again shows the flexibility of the "dot join" notation. In particular, it explains the highly successful "navigation style" of Alloy's syntax, typical of object-oriented (OO) programming.

4.6 Ordering Signatures

The analogy with object-oriented syntax carries further to the hierarchical definition of data types, similar to that of *classes* in OO languages. Given signature A, writing

sig B extends A

means $B \subseteq A$ [18]. Several extensions to the same signature ensure subset disjointness. On the other hand, prefixing keyword **sig** by keyword **abstract** ensures that a signature has no elements except those belonging to its extensions. So,

abstract sig C {}
sig A extends C {}
sig B extends C {}

declares $C = A \cup B$ with $A \cap B = \emptyset$.

This device will be helpful in structuring several refinement steps in Section 8.

5 Verification Theory in Brief

According to the standard theory, there are two kinds of proof to be considered in the design of software from abstract models. One is known as *satisfiability* [26]: For every operation Op specified by a **pre/post** pair whose input is taken from type A and output from type B, proof obligation

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

should be discharged. In case Op is deterministic and performed uniformly over a state space ($A = B$), this PO shrinks to

$$\forall a : a \in A \wedge \text{pre-}Op(a) \Rightarrow Op(a) \in A. \tag{26}$$

Because $a \in A$ and $Op(a) \in A$ check for the state invariant (constraint) before and after Op takes place, (26) is also referred to as *invariant preservation* [26] or *extended static checking* [14].

The second kind of PO has to do with *refinement* steps. In this case, an operation such as Op above is implemented by another operation COp (where "C" stands for *concrete*) running on a strengthened, or more detailed, state space CA [27], glued to the abstract space A by an *abstraction invariant* (ai): For all $x, x' \in A$ and $y, y' \in CA$:

$$\begin{aligned} &ai(x, y) \wedge \text{post-}COp(y', y) \Rightarrow \\ &\quad \exists\, x' : \text{post-}Op(x', x) \wedge ai(x', y'), \end{aligned}$$

should hold. In words: To prove the intended refinement it is necessary to show that once running the refined operation COp in a state (y) reachable from a given abstract state (x), there exists an after-state of Op (x') which abstracts the after-state (y') of COp.

Typically, $ai(x, y)$ will be decomposed into two parts: an abstraction function $x = af(y)$ and a concrete invariant $ci(y)$ imposing constraints at low level. In this situation, which covers the examples given in this paper, the above instantiates to

$$\begin{aligned} &ci(y) \wedge \text{post-}COp(y', y) \Rightarrow \\ &\quad \text{post-}Op(af(y'), af(y)) \wedge ci(y'). \end{aligned} \tag{27}$$

9. The relational generalization of this combinator is termed *split* in [8] and *fork* in [24].
10. See [12]. Exponential type X^A is the set of all functions of type $A \rightarrow X$. This result is implicit in the way n-ary relations are converted into binary relations in the semantics given in [24].
11. See [18] for details on the syntax of Alloy predicates, which are not relevant here.

Moreover, any new operation invented at low level should remain "invisible" at high level, that is, it should refine the identity on states, often termed $Skip$, and such that post-$Skip(x',\ x) \triangleq x' = x$. As can be drawn from (27) by making $Op = Skip$, this boils down to showing that COp preserves concrete invariant ci and that af cannot distinguish y' from y.

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 [27] for details.)

Discharging satisfiability proofs of shape (26) in the AoP is dealt with in [14]. For a related, more tool-oriented (but still "Alloy-centric") approach see, e.g., [17]. Concerning refinement proofs, we will follow the life-cycle of Fig. 1: 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 will not be finished until all calculations are over.

We proceed to the presentation of our case study, recalling that satisfiability at the abstract level was previously proven for the *delete* [14] and *open* [17] operations.

6 Verified File System (VFS) Challenge

There is a healthy trend in formal methods research driven by the idea of a *Grand Challenge* (GC) [28] expected to deliver *"a comprehensive and unified theory of programming"* and *"a repository of verified software"* [28, Section 2]. Mondex [29] was the first GC pilot project. Later, Joshi and Holzmann [16] put up a 2-3 year mini-challenge on verifying a reliable FLASH file-system (VFS) for in-flight software.

An account of the community's response to the VFS mini-challenge is given in Section 10. Further to previous work reported in [17], we have chosen as a case study for the current paper to focus on the refinement steps which introduce the *journaling* mechanism. This is usually found in modern file systems to allow for higher performance and reliability in the face of power loss or unexpected device removal. Moreover, in FLASH devices it contributes to wear leveling of its blocks. This is a nonfunctional requirement intended to prolong the service life of erasable storage media in general.

The inspiration for investing on such a facet of the overall FLASH refinement process comes from work by Schierl et al. [30] on the formalization of UBIFS file system for FLASH memory. We find this aspect of the refinement interesting because it shows the role of properties induced in the refined model by nonfunctional requirements.

This choice reflects one of the major needs in software design, that of *separation of concerns*, that is, the need to factor complex designs into "orthogonal" subdesigns. Journaling is the one (in fact, central) aspect of the design of a FLASH file-system considered in this paper.

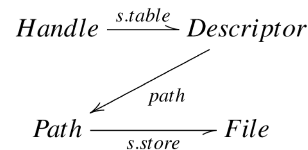
In this setting, we select the refinement of the *delete* operation, already specified and verified in [14], down to the journaled level. This is perhaps the most interesting part of the refinement for its counterintuitive behavior induced by the wear-leveling requirement, as explained in the sequel.

7 Relational Model of a Journaled (Flash) File System

In a typical file system, the file store's data and metadata are stored in the device that hosts the file system. In order to increase performance, some metadata 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, central memory metadata 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* [30]. Altogether, the journal and the remaining metadata stored on the device should make it possible to rebuild the metadata that were stored in central memory as it was before the fault occurred.

While journaling and decentralized metadata 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 refined model presented in the remainder of this section.



As an aside, we recall the main relations of the idealized file system of Section 2 in its Alloy version. This model, which served our purposes to introduce relational modeling and Alloy, is, however, too high-level for *journaling* to be understood and formalized.

Looking at UBIFS [30], for instance, we see that POSIX-compliant file system implementations are based on *nodes* that are stored in volumes. A volume (e.g., a FLASH drive) simply acts as an *array* of nodes that gets written to and read from. This means that the abstractions used thus far to model a file system somehow have to be encoded in such arrays of nodes. To this end, there are several types of nodes, of which we list the ones that are relevant in this context:

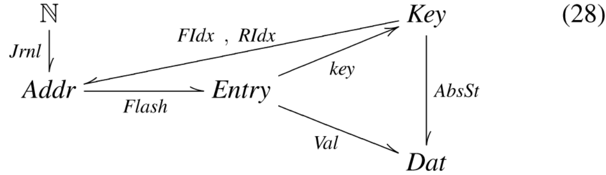
- `inode` (index) nodes that hold file metadata (e.g., size, number of links etc.);
- `data` nodes that hold file contents (as chunks of data);
- `dentry` (directory entry) nodes that record the contents of directories (file names, etc.).

Nodes have *keys* associated with them that enable node retrieval and linking, thus making it possible, without further information, to reconstruct the abstract (tree-based) view one has of a file system simply by sweeping such arrays of nodes. (This is actually what the operating system does when the file system in a volume is checked for consistency.)

Refining our starting, idealized model into volume-level arrays of nodes is surely a very important step. As this has already been dealt with elsewhere (see, e.g., [31]), we have rather focused on another refinement step, this one carried out over the arrays of nodes themselves: *journaling*.

7.1 Journaling

Reasoning about such a low-level concept calls for a context switch from the high-level mental model of a file system to its actual implementation in terms of arrays of nodes. In spite of the semantic gap, both levels are captured by simple relations, of type $Path \rightharpoonup File$ at high-level (as seen above) and of type $Key \rightharpoonup Dat$ at volume level.



(28)

In the sequel we will focus on such a $Key \rightharpoonup Dat$ volume level array which, at this level of refinement, is regarded as the *abstract* structure to be further refined. Diagram (28) above shows how this array, hereinafter referred to as the *abstract store* (*AbsSt*), unfolds into four data structures—labeled $FIdx$, $Jrnl$, $RIdx$, and $Flash$ in the diagram once refined down to (journaled) FLASH level. In detail:

- $Flash$ (FLASH store) is kept in the device and maps memory *addresses* in the data type $Addr$ to array entries ($Entry$) which record keys (Key) and data (Dat). Function key tells which key is stored in each particular array entry. Simple relation Val tells which entries hold data. Entries holding a key and not holding data are regarded as *deleted*, as will be explained shortly.
- Structures $FIdx$ (FLASH index) and $Jrnl$ (journal) hold in-device metadata, whereas $RIdx$ (RAM index) keeps similar data in central memory.
- The RAM index ($RIdx$) provides for fast indexing, mapping each key in Key to the address that currently holds its data in $Flash$; so, by chaining $RIdx$ and $Flash$ one should be able to rebuild the information kept in $AbsSt$. But there is some redundancy in the refinement, as $Flash$ also keeps the converse relationship between addresses ($Addr$) and keys (Key)—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 $RIdx$ are saved in $FIdx$.
- Abnormal operation (power loss or abrupt device removal) calls for the help of journal $Jrnl$, which keeps the list of addresses which have been created since the last commit operation. This means that, in such situations, a *replay* process should take place so as to rebuild $RIdx$ (volatile) from the in-device triple $(Flash, FIdx, Jrnl)$.
- Finally, note the fact that, in diagram (28), list $Jrnl$ is modeled as an association of natural numbers ($\mathbb{N}$) to addresses ($Addr$) indicating the position of each address in the list.

7.2 How It Works

The four structures of the refined state space are suggestively recorded in Fig. 2, extracted verbatim from [30]. The leftmost column of the central array depicts addresses ($Addr$) and the other columns depict keys (Key) and data (Dat), the components of node entries ($Entry$).

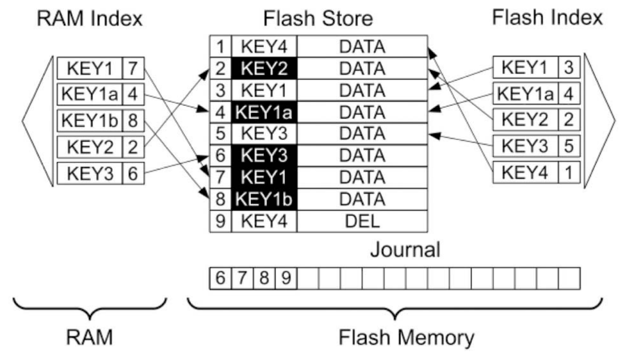


Fig. 2. Journaled FLASH store data structures (picture quoted from [30]).

To ensure wear leveling, cell overwriting is avoided as much as possible. Instead, new addresses are used for updated data values. In the picture, a cell on black background indicates the most recent value associated to a particular key. For instance, $KEY3$ at address 5 is outdated (white background) because it has been updated at address 6 (black background).

The RAM index is, of course, always pointing to the most recent updates. Fig. 2 also shows an outdated FLASH index (still pointing at address 5) in contrast to the updated RAM index (already pointing at address 6).

It is visible that the update of $KEY3$ was the first change taking place upon the last time RAM and FLASH indexes were synchronized, as address 5 is the last in the FLASH index and address 6 is the first entry in the journal.

The journal states the story of what happened afterward. Its second position (address 7) corresponds to the update of $KEY1$. The next two positions in the journal (addresses 8 and 9) reveal that two other updates took place, the last one being the deletion of entry $KEY4$. This was the first node to be stored, at address 1. Instead of freeing this address, a new entry is created (address 9) where $KEY4$ is marked as deleted.

In other words, to delete n cells in $Flash$ one must have n extra free cells. Should this not be the case, a garbage collection operation has to take place, reclaiming all redundant entries such as $KEY4$ in address 1. Lack of space persisting, the device will be full.

7.3 Alloy Encoding

Prior to the formalization of this journaling refinement step, let us see how the relations of diagram (28) are encoded as Alloy signatures, intended for model checking. While abstract node stores ($AbsSt$) are modeled as Alloy maps from keys (Key) to data (Dat):

```
sig AbstSt {r : Key → lone Dat}
```

concrete stores are captured by the four-tuple signature

```
sig CncSt {
  Jrnl : Num → lone Addr,
  Flash : Addr → lone Entry,
  FIdx : Key → lone Addr,
  RIdx : Key → lone Addr
}
```

where each entry in the FLASH store

```
sig Entry {key : one Key, Val : lone Dat}
```

can be regarded as a pair whose right element is of *optional* type $loneD$ [18] in order to model deletion.

8 Refinement Process

The refinement of $AbsSt$ into $CncSt$ will be performed in a stepwise manner so as to introduce concrete details in the place where they are required. There are essentially three stages in the refinement.

1. Introduces the FLASH store accessed by the RAM index; too simplistic, it fails to recover the RAM index in the event of power-loss and ignores the wear-leveling requirement.
2. Adds the Journal to the two previous structures to remedy the downsides of the previous step; the new structure, however, grows indefinitely long.
3. Finally, adds the FLASH index to cache the RAM index and (periodically) clear the Journal.

Among the CRUD[12] $AbsSt$ operations, the *delete* operation—in Alloy:

```
pred absDelete[s: set Key, r',r: Key → lone Dat]
  {r' = (Key-s) <: r}
```

— is selected for refinement because its implementation ends up far more elaborate than first thought, revealing the impact of the wear-leveling and power-loss recovery nonfunctional requirements.

8.1 Refinement—Stage 1

As a starting point we will consider a minimum concrete state model made of the FLASH store and RAM index only, that is,

```
sig CncSt1{
  Flash : Addr → lone Entry,
  RIdx : Key → lone Addr
}
```

in Alloy. The corresponding abstraction invariant (ai_1) will glue abstract states—cf. $AbsSt$ in diagram (28)—to concrete states $CncSt1$—cf. the pair $(Flash, RIdx)$ in the same diagram.

As anticipated by refinement equation (27), we partition ai_1 in terms of an abstraction function af_1 and a concrete invariant ci_1.

8.1.1 Abstraction Function

This is straightforward to define as

$$af_1(Flash, RIdx) \triangleq (active\ Flash) \cdot RIdx, \tag{29}$$

where relation

$$Addr \xrightarrow{active\ Flash} Dat \triangleq Val \cdot Flash \tag{30}$$

captures the $Addr$ to Dat relationship recorded in $Flash$, ignoring all addresses which lead to deleted entries. Chaining this with $RIdx$ yields the Key to Dat map, which is accessible via the RAM index.

The Alloy encoding of definitions (29) and (30) follows, where $cs \in CncSt1$ stands for *concrete state*. Note how "dot join" notation alone does the whole job:

```
fun af1[cs: CncSt1] : Key → Dat {(cs.RIdx).(cs.Flash.active)}
fun active[x: CncSt1.Flash] : Addr → Dat {x.Val}
```

8.1.2 Concrete Invariant

Recall that the abstract state space is inhabited by *simple* relations: No two different data items are associated to the same key. So, for af_1 to be properly typed, it must always deliver one such relation $AbsSt$. Since composition preserves simplicity, $active\ Flash$ will be simple. In the same vein, $(active\ Flash) \cdot RIdx$ will be simple because $RIdx$ is simple. (Note how elementary properties of relation algebra help in type checking the diagram "on the fly.")

We conclude that the simplicity of $RIdx$ and $Flash$ are the first requirements of concrete invariant ci_1 to consider. But we do not need to write these explicitly since they are implicit in the use of "harpoon arrows" in type diagram (28) and keyword $lone$ in Alloy signatures.

8.1.3 Operation Refinement

Proceeding to the refinement process properly said, we want abstract operations over abstract states (relations of type $AbsSt = Key \rightharpoonup Dat$) to be implemented at low level, running on top of concrete states as depicted in diagram (28). Focusing on the delete operation, already specified in Alloy, we can use the semantic rules of Section 4 to derive

$$\begin{aligned} &\text{post-}absDelete(S, AbsSt', AbsSt) \triangleq \\ &\quad AbsSt' = AbsSt \cdot \Phi_{(\notin S)}, \end{aligned} \tag{31}$$

where argument set S tells which keys are to be deleted, and $\Phi_{(\notin S)}$ is a *filter*—the *coreflexive* relation associated to predicate $x \notin S$, produced by semantic rule (22).

How does one implement this operation at FLASH level? At first sight, performing a similar domain-subtract operation on the RAM index $RIdx$,

$$RIdx' = RIdx \cdot \Phi_{(\notin S)}, \tag{32}$$

and leaving $Flash$ unchanged would do since the smaller $RIdx$ is the smaller the abstract state delivered by af_1 (28). And it does, in fact, as the corresponding instance of refinement PO (27)

$$\begin{aligned} &RIdx' = RIdx \cdot \Phi_{(\notin S)} \wedge Flash' = Flash \Rightarrow \\ &\quad af_1(Flash', RIdx') = af_1(Flash, RIdx) \cdot \Phi_{(\notin S)}, \end{aligned} \tag{33}$$

shows, once $RIdx'$ and $Flash'$ are substituted by equals,

$$\begin{aligned} & af_1(Flash, RIdx \cdot \Phi_{(\notin S)}) = af_1(Flash, RIdx) \cdot \Phi_{(\notin S)} \\ &\quad \equiv \{ (29) \} \\ &\qquad active\ Flash \cdot (RIdx \cdot \Phi_{(\notin S)}) \\ &\quad = (active\ Flash \cdot RIdx) \cdot \Phi_{(\notin S)} \\ &\quad \equiv \{\text{composition is associative (75)}\} \\ &\qquad \text{TRUE}. \end{aligned}$$

Why are things not so easy in practice? Note that, for every key deleted in $RIdx$ there is an address in $Flash$ which becomes available for further writing. And further

12. Acronym CRUD stands for *C*reate, *R*ead, *U*pdate, *D*elete.

delete/write cycles may write to such an address over and over again, thus contradicting *wear leveling*.

On the other hand, in the event of a power loss $RIdx$ will be lost, for it lives in volatile RAM. This could in part be remedied by exploiting the redundancy of $Flash$ (28) and running an operation able to recover the Key to $Addr$ association it keeps:

$$Key \xrightarrow{index\ Flash} Addr \triangleq (key \cdot Flash)^\circ, \tag{34}$$

—that is,

fun index[x: CncSt1.Flash] : Key → Addr {(x.key)}

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

$$RIdx \subseteq index\ Flash. \tag{35}$$

Note, however, that (35) cannot be strengthened to an equality, for $index\ Flash$ is not simple in general: By construction, it keeps track of all addresses which have been used to record data for a given key. Besides, information is missing about which addresses correspond to the most recent updates. So, $RIdx$ is not recoverable at all.

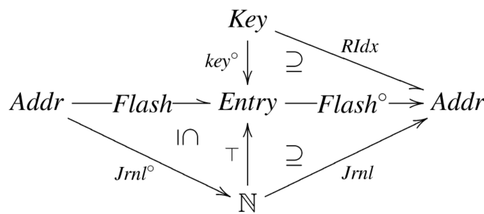
This leads to a revision of the model, which brings *journal* $Jrnl$ into the scene, intended to keep the order in which addresses are created.

8.2 Refinement—Stage 2

The revision consists, first of all, in adding the journal (another simple relation) to the model

```
sig CncSt2 extends CncSt1 {
 Jrnl : Num → lone Addr
}
```

and then in drafting the new concrete invariant (ci_2) in a diagram:



The top-right triangle records (35); the other two triangles capture referential integrity and together ensure that the addresses logged in $Jrnl$ are exactly those found in $Flash$. By resorting to the algebra of the ρ and δ relational operators mentioned in Section 3, we put ci_2 in symbols as follows:[13]

$$ci_2(Flash, RIdx, Jrnl) \triangleq RIdx \subseteq index\ Flash \wedge \rho Jrnl = \delta Flash. \tag{36}$$

A consequence of (36) worth mentioning is the *injectivity* of $RIdx$. This arises from the simplicity of $key \cdot Flash$ and the rules of thumb of Section 3: The converse of simple is injective, etc. Moreover,

13. Since ci_1 is implicit in the overall typing, we do not quote it in ci_2.

$$\rho(index\ Flash) = \delta Flash, \tag{37}$$

as can be easily checked using the algebra of domain and range—see, e.g., (88) and (93) in Appendix A.2. (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_2, rendering $RIdx$ functionally dependent on the other components of the concrete state:

$$RIdx = replay(Flash, Jrnl), \tag{38}$$

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

8.2.1 The *replay* Function

The semantics of *replay* are based on handling relation $index\ Flash$ (34), in two steps.

- First, for each key in $index\ Flash$ select the address which holds its most recent update (this will yield a simple relation from Key to $Addr$).
- Second, filter deleted keys out.

Below we give a relational definition of the *replay* function in which these two steps are captured by dedicated relational combinators. One of these in particular shows how relational algebra scales up by definition of "clever" operators which enable one to deal with inductive problems in a noninductive fashion.

First, we need to define an ordering on addresses, taking into account their position in the journal:

$$a \geq_{Jrnl} b \equiv \exists\, i, j : a\ Jrnl\ i \wedge b\ Jrnl\ j \wedge\ i \geq j. \tag{39}$$

This order on addresses works by comparing their relative positions in $Jrnl$, larger positions meaning more recent updates. In relation algebra notation, (39) shrinks to

$$Addr \xleftarrow{\geq_{Jrnl}} Addr \triangleq Jrnl \cdot \geq \cdot Jrnl^\circ. \tag{40}$$

Finally, the relational specification of *replay* relies on this ordering

$$replay(Flash, Jrnl) \triangleq active\ Flash \triangleleft (index\ Flash \upharpoonleft (\geq_{Jrnl})), \tag{41}$$

and in two newly defined binary relational combinators ($\triangleleft$ and $\upharpoonleft$) which are explained in the following paragraphs.

8.2.2 The $\triangleleft$ (Postrestrict) Combinator

This combinator, defined by

$$S \triangleleft R \triangleq \delta S \cdot R, \tag{42}$$

(read $S \triangleleft R$ as "R wherever S is defined") postrestricts a given relation R by the domain of some other relation S. Clearly, properties

$$(S \triangleleft R) \cdot T = S \triangleleft (R \cdot T), \tag{43}$$

$$S \triangleleft (R \cup V) = (S \triangleleft R) \cup (S \triangleleft V), \tag{44}$$

hold. Combinator (42) is used in (41) in the second step, filtering deleted keys out (i.e., those not present in

active Flash). It is encoded in Alloy as function $ifdef$, given in Appendix A.1.

8.2.3 The $\upharpoonright$ (Shrinking) Combinator

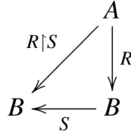
Let relations R and S be typed as in the diagram aside. We will write the expression $R \upharpoonright S$ to denote the effect of *shrinking* R by S, that is, of converting R into a smaller, simple relation by looking at particular (e.g., maximal) elements of its range relative to S. This is captured by definition

$$R \upharpoonright S \triangleq R \cap S/R^{\circ}$$

and the corresponding universal property

$$X \subseteq R \upharpoonright S \equiv X \subseteq R \wedge X \cdot R^{\circ} \subseteq S, \tag{45}$$

which ensure that $R \upharpoonright S$ is the largest subrelation X of R such that, for all $b', b \in B$, if there exists some $a \in A$ such that $b' X a \wedge b R a$ holds, then $b' S b$ holds.



This combinator is a generalization of the max operator of the AoP [8]. Its pointwise meaning is captured in Alloy by function *shrinkby*, given in Appendix A.1.

Let, for instance, S be id (the equality relation) and R be simple in (45). Then, since $R \cdot R^{\circ} \subseteq id$ holds (5), the maximal solution of (45) 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:

$$(R \upharpoonright S) \cdot \Phi = (R \cdot \Phi) \upharpoonright S, \tag{46}$$

$$R \upharpoonright S = R \upharpoonright (\rho R \cdot S \cdot \rho R), \tag{47}$$

$$\mathrm{R} \upharpoonright \mathrm{S} \text{ is simple} \Leftarrow \mathrm{S} \text{ is antisymmetric}, \tag{48}$$

$$(R \cup S) \upharpoonright U = (R \upharpoonright U) \cap U/S^{\circ} \cup (S \upharpoonright U) \cap U/R^{\circ}, \tag{49}$$

as well as a corollary of (49)

$$(R \cup S) \upharpoonright U = (R \upharpoonright U) \cup (S \upharpoonright U) \Leftarrow R \cdot S^{\circ} \subseteq \bot, \tag{50}$$

all relevant in the sequel. For instance, thanks to (48), the *shrinking* combinator *"simplifies"* $index\ Flash$ in (41) via relation (40).

However, to ensure (40) antisymmetric, simple $Jrnl$ also needs to be injective so as to prevent address duplication:

$$\geq_{Jrnl} \cap \geq_{Jrnl}{}^{\circ} \subseteq id \Leftarrow \mathrm{Jrnl} \text{ simple and injective}. \tag{51}$$

The proof of (51) can be found in Appendix A.2.

8.2.4 Final Touch

It follows from (41) that clause (38) strengthens (35), since both $R \upharpoonright S$ and $S \triangleleft R$ are subrelations of R, in general. So (38) replaces (35) in the final version of concrete invariant ci_2, which also records the injectivity of $Jrnl$:

$$\begin{aligned} ci_2(Flash, RIdx, Jrnl) \triangleq \\ & RIdx = replay(Flash, Jrnl) \wedge \\ & \rho Jrnl = \delta Flash \wedge \\ & Jrnl \text{ injective}. \end{aligned} \tag{52}$$

Encoded in Alloy, this invariant becomes

```
pred ci2[cs : CncSt2] {
  cs.RIdx = cs.replay
  cs.Jrnl.ran = cs.Flash.dom
  injective[cs.Jrnl,Num]
}
```

where $injective$ is a function available from Alloy module $relation$ and—last but not least—the $replay$ function (41) is given by

```
fun replay[cs: CncSt2] : Key Addr {
  let geq =* (ordering/prev),
      geqj = ~(cs.Jrnl).geq.(cs.Jrnl),
      ix = shrinkby[cs.Flash.index, geqj] |
           ifdef [cs.Flash.active,ix]
}
```

8.2.5 Revised Delete Operation

As happens with the $replay$ function, the revised delete operation is better explained in its three 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 (32), 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 postcondition, as follows:

Let $Key \xrightarrow{N} Addr \xleftarrow{M} \mathbb{N}$ *be two simple and injective relations, where* N *associates fresh* FLASH *store addresses to the keys to be deleted (specified in a given set* S*) and* M *orders such addresses in an underspecified way. Then,*

$$RIdx' = RIdx \cdot \Phi_{(\notin S)}, \tag{53}$$

$$Flash' = Flash \cup delN, \tag{54}$$

$$Jrnl' = Jrnl \cup M, \tag{55}$$

under the following conditions: 1) addresses in N *are fresh,*

$$Flash \cdot N = \bot, \tag{56}$$

and precisely those listed in M*:*

$$\rho N = \rho M. \tag{57}$$

2) The keys in N *are exactly those to be deleted:*

$$\delta N = \Phi_{(\in S)} \cdot \delta RIdx. \tag{58}$$

3) Positions in M are such that $Jrnl \cup M$ appends M to $Jrnl$:

$$M^\circ \cdot \top \cdot Jrnl \subseteq >. \tag{59}$$

4) Finally, relation $Entry \xleftarrow{delN} Addr$ records $Key \xrightarrow{N} Addr$ into the FLASH *store associating to each fresh address its key and no data, meaning a deletion. Thus, the following requirements:*

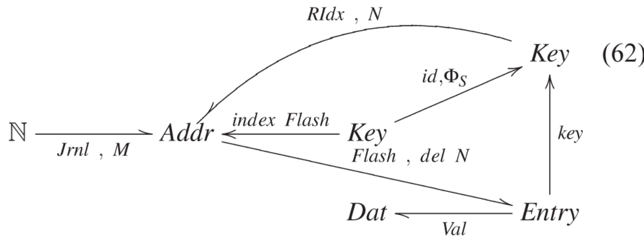
$$key \cdot (delN) = N^\circ \tag{60}$$

(the whole N is recoverable from $delN$);

$$Val \cdot (del\ N) = \bot \tag{61}$$

(no data in $delN$ entries, so as to mark keys as deleted.)

The diagram which follows helps in type checking the postcondition just given: the synthesis of the Alloy for $delN$




```
fun del[n: Key → Addr] : Addr → Entry {
    { a: n.ran, e: Entry | a in e.key.n and no e.Val}
}
```

—equivalent to $(del\ N)^\circ = N \cdot key \cap \bot / Val^\circ$ arising from properties (60) and (61)—is an interesting exercise in Alloy-meets-AoP interfacing, omitted for space economy. Note how the negative part of the Alloy definition is captured by relational division (13). Another way to express this is to resort to the complemented domain operator from (97) and (98), whereby closed formula $delN = \overline{\delta Val} \cdot (N \cdot key)^\circ$ is easy to obtain.

Note that, for diagram (62) to type-check, we still need to prove that arrows $Flash'$ and $Jrnl'$ are simple,

$$Flash' = Flash \cup del\ N \text{ is simple,} \tag{63}$$

$$Jrnl' = Jrnl \cup M \text{ is simple,} \tag{64}$$

as $RIdx'$ is trivially so. The calculations of (63) and (64) can be found in Appendix A.2.

A number of facts are useful recording at this point for later use.

- All fresh entries in $Flash'$ are inactive:

  $$active(del\ N) = \bot. \tag{65}$$

  This follows immediately from (30) and (61).
- $index$ recovers N from $del\ N$:

  $$index(del\ N) = N. \tag{66}$$

  This follows immediately from (34) and (60).
- $Jrnl'$ adds nothing to the update of $index\ Flash$:

  $$index\ Flash \upharpoonright (\geq_{Jrnl'}) = index\ Flash \upharpoonright (\geq_{Jrnl}). \tag{67}$$

  (Proof in Appendix A.2.)

Finally, the whole postcondition is transliterated to Alloy:

```
pred Delete[cs,cs': CncSt2, s: set Key] {
  some n: Key → lone Addr, m: Num → lone Addr {
    injective[n, Key] and injective[m, Key]
    no n.(cs.Flash)
    n.ran = m.ran
    n.dom = (s & cs.RIdx.dom)
    cs.Jrnl.(Addr→ Addr).(~m) in ^(ordering/prev)
    cs'.Jrnl = cs.Jrnl + m
    cs'.Flash = cs.Flash + n.del
    cs'.RIdx = (cs.RIdx.dom-s) <: (cs.RIdx)
  }
}
```

8.2.6 Checking the Proposed Refinement

It is handy to split proof obligation (27) in its two components, one dealing with concrete invariant maintenance,

$$ci(y) \wedge \text{post-}COp(y', y) \;\Rightarrow\; ci(y'), \tag{68}$$

and the other ensuring safe refinement,

$$ci(y) \wedge \text{post-}COp(y', y) \;\Rightarrow\; \text{post-}Op(af(y'), af(y)). \tag{69}$$

The instance of (69) for the $Delete$ operation is, in Alloy's assertion syntax:

```
assert po69 {
  all cs,cs': CncSt2, s : set Key |
    (ci2[cs] and Delete[cs,cs',s]) ⇒
              absDelete[s,cs'.af1,cs.af1]
}
```

The instance of (68) for $ci = ci_2$ splits into

$$ci_2(Flash, RIdx, Jrnl) \wedge \text{clauses}(53-61); \Rightarrow \rho Jrnl' = \delta Flash', \tag{70}$$

$$ci_2(Flash, RIdx, Jrnl) \wedge \text{clauses}(53-61) \Rightarrow Jrnl' \text{ injective}, \tag{71}$$

$$ci_2(Flash, RIdx, Jrnl) \wedge \text{clauses}(53-61) \;\Rightarrow\; RIdx' = replay(Flash', Jrnl'), \tag{72}$$

giving rise to the Alloy assertions:

```
assert po70 {
  all cs,cs': CncSt2, s : set Key |
  (ci2[cs] and Delete[cs,cs',s]) ⇒
            cs'.Jrnl.ran = cs'.Flash.dom
}
assert po71 {
  all cs,cs': CncSt2, s : set Key |
  (ci2[cs] and Delete[cs,cs',s]) ⇒ injective[cs'.Jrnl,Num]
}
assert po72 {
  all cs,cs': CncSt2, s : set Key |
  (ci2[cs] and Delete[cs,cs',s]) ⇒ cs'.RIdx = cs'.replay
}
```

Calculation of (69) proceeds by rewriting the consequent of the implication, assuming the antecedent and all the substitutions implicit:

$$
\begin{aligned}
& af(Flash', RIdx', Jrnl') = af(Flash, RIdx, Jrnl) \cdot \Phi_{(\notin S)} \\
\equiv & \{ \text{ (29) twice } \} \\
& active\ Flash' \cdot RIdx' = (active\ Flash \cdot RIdx) \cdot \Phi_{(\notin S)} \\
\equiv & \{ (53)\ ;\ (54) \ \} \\
& active(Flash \cup del\ N) \cdot RIdx' = active\ Flash \cdot RIdx' \\
\Leftarrow & \{ \text{Leibniz} \} \\
& active(Flash \cup delN) = active\ Flash \\
\equiv & \{ \ active(30) \text{ distributes through union} \} \\
& active\ Flash \cup active(del\ N) = active\ Flash \\
\equiv & \{ (65) \} \\
& active\ Flash = active\ Flash.
\end{aligned}
$$

Not as immediate as calculation (33) performed in refinement step 1, refinement PO (69) for $Delete$ was easy to discharge after all. So, the crux of the refinement step must reside elsewhere: in fact, in preserving the concrete invariant's clauses (70), (57), and (72), which ensure referential integrity and power loss recovery, as we shall see shortly.

The first two such POs to check, (70) and (71), are still easy exercises in the AoP (see Appendix A.2). By contrast, the calculation of PO (72) 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 leveling* requirement. It proceeds by rewriting one side of the target equality ($replay(Flash', Jrnl')$) into the other ($RIdx'$), under the given context. We abbreviate $\geq_{Jrnl'}$ to $\sqsupseteq$ for improved readability.

$$
\begin{aligned}
& replay(Flash', Jrnl') \\
= & \{ \text{ substitutions enabled by clauses (55) to (61)} \} \\
& replay(Flash \cup del\ N, Jrnl') \\
= & \{ \text{ definition (41) ; active distributes over union } \} \\
& (active\ Flash \cup active(del\ N)) \lhd \\
& \quad (index(Flash \cup del\ N) \upharpoonright \sqsupseteq) \\
= & \{ \text{ (65); index distributes over union; del (66) } \} \\
& active\ Flash \lhd ((index\ Flash \cup N) \upharpoonright \sqsupseteq) \\
= & \{ \text{ splitting index Flash in two disjoint parts } \} \\
& active\ Flash \lhd \\
& ((index\ Flash \cdot \overline{\delta N} \cup index\ Flash \cdot \delta N \cup N) \upharpoonright \sqsupseteq) \\
= & \{ \text{ distributions (50) and (44) } \} \\
& (active\ Flash \lhd ((index\ Flash \cdot \overline{\delta N})) \upharpoonright \sqsupseteq) \quad \cup \\
& (active\ Flash \lhd ((index\ Flash \cdot \delta N \cup N) \upharpoonright \sqsupseteq)) \\
= & \{ \text{ (46); (67) } \} \\
& (active\ Flash\ \lhd (index\ Flash \upharpoonright (\geq_{Jrnl})) \cdot \overline{\delta N}) \quad \cup \\
& (active\ Flash\ \lhd ((index\ Flash \cdot \delta N \cup N) \upharpoonright \sqsupseteq)) \\
= & \{ \ (index\ Flash \cdot \delta N \cup N) \upharpoonright \sqsupseteq is N, \text{ see (73) below } \} \\
& replay(Flash, Jrnl) \cdot \overline{\delta N} \cup active\ Flash \lhd N \\
= & \{ \text{ definition (52) ; } \delta(active\ Flash) \subseteq \ \delta\ Flash; (56) \ \} \\
& RIdx \cdot \overline{\delta N} \cup \bot \\
= & \{ \text{ (77); clause (58); (99) } \} \\
& RIdx \cdot \overline{\Phi_{(\in S)}} \\
= & \{ \text{ (100); clause (53) } \} \\
& RIdx'.
\end{aligned}
$$

The intuition behind the step left unjustified in the calculation above,

$$
(index\ Flash \cdot \delta N \cup N) \upharpoonright (\geq_{Jrnl'}) = N \upharpoonright (\geq_{Jrnl'}) = N, \tag{73}
$$

is that, because all addresses in N are greater than those in $Flash$ (as granted by M in $Jrnl'$), all entries in $index\ Flash \cdot \delta N$ will be overwritten by N for they are all bound to conflict with N.

This provides an interesting opportunity for reflecting on the overall *Alloy-meets-AoP* round-trip philosophy. It turned out that, at the time of calculating the above proof obligation, conjecture (73) was made, offering a significant simplification of the argument and promising its conclusion. What we did was to interrupt the proof and model-check (73) as the pair of Alloy assertions:

```
assert po73a {
  all cs,cs,: CncSt2, s : set Key,
      n: Key → lone Addr, m: Num → lone Addr |
        po73_context[cs,cs',s,n,m] ⇒
          let geq = *(ordering/prev) |
            let ord = ~(cs'.Jrnl).geq.(cs'.Jrnl) |
                shrinkby[(n.dom) <: (cs.Flash.index) +
                        n, ord] = shrinkby[n, ord]
}
```

and

```
assert po73b {
  all cs,cs': CncSt2, s : set Key,
      n: Key → lone Addr, m: Num → lone Addr |
        po73_context[cs,cs',s,n,m] =>
          let geq = *(ordering/prev) |
            let ord = ~(cs'.Jrnl).geq.(cs'.Jrnl) |
                shrinkby[n, ord] = n
}
```

under the proof's context:

```
pred po73_context
  [cs,cs': CncSt2, s : set Key,
    n: Key → lone Addr,
    m: Num → lone Addr] {
    ci2[cs]
    injective[n, Key]
    injective[m, Key]
    no n.(cs.Flash)
    n.ran = m.ran
    n.dom = s & cs.RIdx.dom
    cs.Jrnl.(Addr→ Addr).(~m) in ^(ordering/prev)
    cs'.Jrnl = cs.Jrnl + m
    cs'.Flash = cs.Flash + n.del
    cs'.RIdx = (cs.RIdx.dom-s) <: (cs.RIdx)
}
```

Not getting any counterexample (given a large enough scope), we trusted the conjecture and completed the proof, as given above. Then, we turned our attention back to

proving (73) itself, by AoP calculation, which was not at all immediate. (Readers interested in the details of this proof can find it in Appendix A.3.) Clearly, the AoP exercise of proving (72) could have been a waste of time should (73) not be true. Thus, the practical role of Alloy model-checking of intermediate results, interweaved in AoP calculations, to reduce the risk of relying on bad conjectures.

8.2.7 Remark on Scoping

One common pitfall when checking assertions with the Alloy Analyzer is to rely on *no counterexamples found* reports when in fact there might be some. The tool has a standard scope that defines how many instances of each type it can create when checking (or instantiating) a model. This scope can, however, be customized. If the scope with which Alloy Analyzer checks a property is too narrow to create instances of the model, it will always report no counterexample found. This does not mean that there are no counterexamples to the property, it only means that the space of instances in which Alloy Analyzer searches for the counterexamples is empty.

To avoid this type of problem, one should constantly check if it is possible to instantiate the model via the Alloy run command. If at some point, with the standard scope size, Alloy Analyzer cannot instantiate the model, we incrementally increase the scope size until it can. However, if we reach a point where the scope size is too large and it is still not possible to instantiate the model, then we know we have introduced some contradiction and need to fix the model (see [31, Section 5.2] for how to identify contradictions in Alloy models). For assessing how large is *too large*, we estimate the minimum scope needed to instantiate the model by analyzing the types defined in the model. For instance, if we partition a signature in four subsignatures (think of an enumerated type) and some constraint makes it imperative that instances of all four subsignatures exist, then it is obvious that the minimum scope size (at least for the partitioned signature) be 4.

9 Last Touch in Refinement Process

A great disadvantage of the refined model presented above is the growth of journal $Jrnl$, which is bound to cover the whole $Flash$ at any time. Power loss recovery of $RIdx$ by the $replay$ function will thus take longer and longer as $Jrnl$ grows. This suggests that, from time to time, $Jrnl$ should be cleared up while saving the contents of $RIdx$ persistently. This is the purpose of $FIdx$ (FLASH index), the last addition to our state model:

```
sig CncSt3 extends CncSt2 {
 FIdx : Key → lone Addr
}
```

In this way, $Jrnl$ will keep only the "difference" between $RIdx$ and its *cache* $FIdx$. Introducing $FIdx$ requires a *commit* operation which basically saves $RIdx$ into $FIdx$ and clears $Jrnl$, as specified by the following postcondition:

$$Jrnl' = \bot,$$
$$FIdx' = RIdx,$$
$$Flash \text{ remains unchanged},$$
$$RIdx \text{ remains unchanged}.$$

As a consequence of caching $RIdx$ into $FIdx$, both concrete invariant ci_2 (52) and the *replay* operation (37) need to be upgraded. Earlier on, $RIdx$ would be rebuilt just by taking journal $Jrnl$ into account. Now $Jrnl$ does not cover the whole $Flash$, only that part changed since the last commit, which can be retrieved from

$$Jrnl^\circ \lhd index\ Flash.$$

The rest can be found in the FLASH index $FIdx$, but it is necessary to override this with the changes meanwhile recorded in $Jrnl$. This has the advantage of replaying only the operations that happened after the last check point (*commit*), as captured by redefinition

$$\begin{array}{l} replay(Flash, FIdx, Jrnl) \ \triangleq \\ \quad active\ Flash \lhd \\ \qquad (FIdx \dagger ((Jrnl^\circ \lhd index\ Flash) \upharpoonright (\geq_{Jrnl}))), \end{array}$$

where † denotes relational overriding [12].

In turn, the concrete invariant calls for further upgrading so as to record extra properties of the state. We omit their formulation and the whole calculation of this extra refinement step from the current paper, which, despite the added complexity, is performed along the same lines and strategy. What is important to know is that it is this step which introduces the FLASH index into the overall design.

10 Related Work

10.1 Blending Proof Checking and Model Checking

The integration of the complementary technologies of model checking and proof checking is a popular subject in the program verification community [32]. In particular, the idea of preceding correctness proofs by model-checking to *"quickly eliminate false conjectures"* can be found in, e.g., [33], where a program verification method that combines random testing, model checking and interactive theorem proving in Agda/Alfa is proposed.

The main difference compared to our work resides in the level at which reasoning takes place. This can be perceived by following the same example, checking the type of the function which doubles a number, as dealt with by Owre et al. [32] and Oliveira [14]: The PVS in the former handles quantified formulæ while the latter performs quantifier-free calculation of weakest preconditions.

The reader is also referred to *Dynamite* [24], a tool that blends Alloy with the semi-automatic theorem prover PVS through the integration of a sound automatic translation of Alloy models to PVS and a complete proof calculus for Alloy based on fork algebra.

Finally, theorem provers such as Prover9 [34] work directly with generalizations of the algebraic structures of the kind one plays with in relational algebra. Some thoughts on linking Alloy to Prover9 can be found in [34].

10.2 Alloy and Refinement Verification

Alloy has been used in conjunction with Event-B [35] and Z [36]. In [35], the Alloy Analyzer is used to validate some invariants for which an automatic proof was not achieved through theorem proving. Bolton [36] reports on how the

tool's simulation capabilities can be of help in verifying data refinement in Z. Instead of using state space search capabilities, Bolton [36] 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 is required to produce instances of the retrieve relation to verify the refinement.

In [31], Alloy alone is used to check that a FLASH file system model conforms to a given POSIX specification. Due to the proximity between our case study and the one of [31], a more in-depth comparison follows.

10.3 Modeling FLASH Devices in Alloy

Kang and Jackson [31] model a FLASH file system to *illustrate key concepts of Alloy*. Two (at first) independent models are developed, one covering a POSIX compliant file system and the other a file system tailored to the specifics of FLASH devices, including memory layout. The highly abstracted POSIX file system is linked to the detailed FLASH file system through an abstraction relation.

The most obvious difference between this work and the current paper's case study is that in this one we do not model a FLASH memory device *as such*. Concerning which file system operations are modeled, the two works are complementary: deletion in our case study, writing and reading in [31]. This alone leads to different levels of detail since deletion is parametric on the contents of what is deleted, in contrast to [31] where such contents are central to the modeling.

Both papers present refinement exercises, with a difference in the number of refinement steps. We incrementally refine a model through a succession of small steps, whereas they produce two independent models and later provide evidence that one conforms to the other.

Another difference concerns which file system features are being addressed. While both papers are concerned with *wear leveling*, the main focus of our paper is the *journaling* feature, which increases performance and assures that volatile data structures can be rebuilt in the presence of faults. The main focus of [31] is the reliability of the write operation, also in the presence of faults; wear leveling is considered as part of the garbage collection operation.

Different operations combined with different file system features addressed lead to the most significant difference: Where we view the FLASH store as an array of opaque entries, they model it as an array of *inodes* and *data nodes*. Our entries encapsulate their inodes and data nodes, but such details are irrelevant as they do not play any role in the verification of the journaling mechanism.

Despite this fundamental difference, the wear leveling mechanisms in both models are fairly similar. In neither model are data erased upon becoming obsolete; instead, modified copies of the existing data are created in free areas of the device to replace outdated ones. Tracking obsolete entries is, however, different: In our model this is part of journaling, not garbage collection.

All-in-all, the model in [31] is more detailed than ours with respect to low-level file system and FLASH device data structures. We rely on a *separation of concerns* which isolates the facet of FLASH operation which we want to analyze: journaling. This is why many details become irrelevant.

10.4 Other Approaches to the VFS Challenge

The VFS challenge has been the focus of several research groups, leading to a vast range of approaches from low level FLASH devices, Linux compliant file systems, to highly abstract file systems models.

One angle of approach to the VFS challenge is the verification of real file system software. Galloway et al. [37] report on the verification of the Linux virtual file system using SPIN and SMART to perform simulation and model checking, respectively. Mühlberg and Lüttgen [38] present a novel verification tool named SOCA that is capable of model checking binary code. This is particularly useful in the verification of the Linux virtual file system, as it is written in C and inlined assembly code. Yang et al. [39] analyze several file system implementations taking journaling into account. Their analysis resulted in several errors being reported back to the developers. Schierl et al. [30] report 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 journaling using our approach.

On a more model-driven approach, Butterfield and Catháin [40] tackle the problem in a bottom up fashion by modeling and verifying FLASH memory devices based on the Open NAND Flash Interface (ONFi) specification. Their work builds from memory layout models in Z [41] to more advanced models in CSP [40] that already take into account the interleaving of concurrent low-level operations.

Other researchers, however, take the top-down approach. Damchoom and Butler [42] use Event-B to model and verify an abstract tree-based file system. Their baseline is a dual notion of refinement: *horizontal* for feature augmentation, and *vertical* for structural refinement. They reach a point where, through machine decomposition in Event-B, it is possible to split the model in the part to be implemented in hardware and that to be implemented in software. Hesselink and Lali [43] present an abstract file system model similar to ours, where the file store is a partial function of paths to data. However, instead of model-checking, this work relies on theorem proving to verify both the abstract model and the refinement steps toward a pointer implementation which contemplates the addition of access control over files in the system.

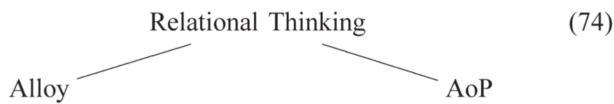
Not all research related to the VFS challenge is about designing new file store models. Some researchers have chosen to take existing models one step further, as is the case of [44] in starting from an existing Z specification by Morgan and Sufrin [45]. The outcome is the mechanization of the required proofs using the Z/Eves theorem prover.

11 Conclusions

11.1 Relational Thinking

The approach to software design advocated in this paper puts together two trends in software validation which usually do not interact with each other—model checking (in Alloy) and program calculation (in the AoP).

Such interaction between two quite different techniques (both in spirit and practice) is made possible under the umbrella of what may be termed "Relational Thinking":



(74)

—a discipline which regards *binary relations* as the main building block of the process of capturing requirements into mathematical models, promising a way to verified code through refinement.

Relational thinking pushes Alloy's lemma "everything is a relation" even further by restricting to binary relations and quantifier-free notation. There is no contradiction here: Binary relation algebra with pairing[14] is as expressive as the algebra of n-ary relations [21]. In addition, binary relations are naturally pictured as arrows in diagrams. In this way, not only types and operations (data types) but also constraints (business rules) can be displayed by semantically rich drawings.

On the theory side, binary relations are the morphisms of the allegory of relations [7] which underlies the AoP [8]; on the practical side, a subset of the Alloy modeling language is *pointfree* enough to accommodate the core combinators of the algebra, notably *dot join*. The "Alloy-meets-AoP" gluing effect is therefore natural and easy to understand and teach. It contrasts, in its surprising simplicity, with the unnecessary convolution of languages, (graphical) notations, and tools currently offered for commercial software design.

In summary, the approach proposes a two-stage analysis that combines incomplete reasoning (Alloy) with complete reasoning (AoP), rather than a two-stage analysis that uses model-theoretic methods followed by proof-theoretic ones (such a distinction is not the key issue in the methodology which is proposed).

Thinking software in this way calls for *abstraction* skills [46], no doubt ingredient number-one in software construction. (In the words of Wing [47], *"computing is the automation of our abstractions."*) It also caters to two complementary aspects of software validation: checking and proving. The most practically inclined are likely to feel comfortable enough within model-checking; others will not trust checking at all and will want to proceed to formal proofs straight away. Clearly, each "leg" of diagram (74) has its own merits and drawbacks. By putting them together we get the best of both worlds.

11.2 Broadening Scope

The history of scientific technology [48] tells us that complex phenomena end up being explained by simple formulæ, leading to endurable knowledge once scientists find not only a suitable *abstract* view of reality but also the *right notation* for expressing it. Convoluted concepts and notations are doomed to fail. They may live for a while, but sooner or later they will be replaced.

Binary relations cannot be a simpler concept. They generalize *graphs*, which have provided endurable support for the software sciences. Depicted as arrows, typed relations make up diagrams obtained just by arrow chaining. Quite often, modeling functionalities or imposing constraints on abstract models amounts to no more than following paths from one place to another in a diagram; recall, e.g., (27). Such paths always exist between any two nodes in a diagram thanks to relation *converse* (an advantage of *relational* versus *functional* thinking).

Binary relations lead to proofs by calculation in the tradition of school algebra, easy to get used to once the laws of the underlying algebra are understood. The fact that AoP calculations are *quantifier* and *induction* free should be emphasized.

Education is surely an issue in relational thinking. There is no fundamental reason for discrete maths not being taught to high school students in terms of binary relations, in the quantifier-free style. Recent, successful experience in solving *puzzles* in Alloy as a way of introducing teenagers to computing [49] suggests our approach as a candidate for implementing *computational thinking* [47].

Relational thinking's focus on relations paves the way for many other models of computation, such as Petri Nets, LTS, FSA, etc. Relational databases and modeling languages such as the UML also rely heavily on relations. However, such relations are n-ary and make reasoning about them difficult and convoluted, full of ellipsis ("...") notation. Oliveira [15] shows that functional and multivalued dependencies in database theory can be calculated using binary relations only, thus following the principles advocated in the current paper.

11.3 Downsides

The coexistence of two notations in (74) has a negative side: It increases the learning burden inherent to the approach. The definition of a shared logic should be considered in the future. This is, however, not easy to achieve as choosing one notation in detriment of the other has pros and cons. Sticking to AoP notation only, for instance, would mean losing model checking; enlarging Alloy to include the full range of operators of the AoP in order to write proof steps in Alloy would mean this losing its light weight, which is so appreciated by users. As the approach is still in its infancy, the proposed Alloy-AoP round-trip is perhaps inevitable at this early stage. Automated translators between AoP and Alloy will therefore be essential.

While Alloy provides an interactive tool that is simple to use, AoP proofs are carried out by manual calculation. These might be more elegant and agile than traditional (pointwise) proofs, however they still require a considerable amount of knowledge, creativity, and, above all, practice. Moreover, there is no tool support for checking proofs from top to bottom, which means that one is not free from subtle (but harmful) mistakes.

This calls for education in calculational techniques, which, unfortunately, cannot be found in most university curricula. Changing these will require, perhaps, the effort of a whole generation. At Minho, relational thinking has been taught to master's students for several years now [50]. This is complemented by later studying automated proof theory and practicing with off-the-shelf theorem provers. The students are often shocked by the declarative flavor of Alloy in the first place, but they soon realize that what they are asked to do is *to think about* software, not to write programs straight away. Maturing a design before implementation is contrary to the dominant culture in agile methods.

Parnas [2] writes: *"good methods, properly explained, sell themselves. Our present methods do not sell beyond the first trial."* Relational thinking is in its infancy. We hope to be able to properly explain, develop, and sell it. Is it a *good*, scalable method? We think so, but only time and experience will tell.

14. Cf. the *fork algebras* of [24].

12 Future Work

12.1 Need for a Body of Knowledge

Experience in formal modeling tells that designs are *repetitive* in the sense of instantiating *generic* constraints whose ubiquity calls for cataloguing and classification.

Taking advantage of relational thinking's *constraints-as-rectangles* approach, we intend to implement the idea of drawing a "constraint bestiary" [51] capturing such "constraint patterns" in a way easy to memorize, enriched with the corresponding proof theories. These should encompass checking all CRUD operations on the constrained relations.

Typical constraint patterns on collections of name spaces are, for instance, referential-integrity, domain (respectively, range) coherence or disjointness, domain-closure with respect to some ordering or function, and so on [51].

12.2 Tuning Alloy for the AoP

Alloy's type system is too liberal for AoP's taste and can be dangerous in places. For instance, during our experiments, we mistyped a relation name for another in a pointfree expression, something an AoP type checker would immediately complain about. Instead, Alloy accepted what we wrote and silently computed an empty "dot join" relation, trivializing the proof obligation and defectively generating no counterexamples.

To prevent situations like this, which jeopardize the purpose of the tool, we plan to develop an external type checker for Alloy adhering to AoP's strict typing rules which we will run as a preprocessor before submitting scripts to the Alloy Analyzer.

12.3 Need for Automation

The "AoP leg" of relational thinking relies on quantifier-free relational algebra reasoning in a style which is reminiscent of the routines of school algebra. Necco et al. [52] experimented with automating such routines. For this, a type-directed, strategic term rewriting system is developed in Haskell, which can be used to simplify relational proof obligations and ultimately reduce them to tautologies. In [52], we have checked how to use such reduction strategies in providing extended static checking for design constraints.

A routine central to equation solving consists of "shunting" expressions from one side to the other of (in)equations, while "changing signs." Galois connections generalize this routine in their exchanging between lower an upper adjoints, e.g., $(\cdot R)$ exchanged by $(/R)$ in (12). Clever use of this tactic is one of the hallmarks of AoP. Silva and Oliveira [53] report on the development of a proof assistant which is solely based on the (higher order) algebra of Galois connections.

We intend to combine these experimental systems with the Alloy tool reported in [25], toward the development of a tool-chain supporting the life-cycle of relational thinking illustrated in this paper. The alternative use of the *Dynamite* proving system [24] for the same purpose will be considered.

12.4 VFS: Need for Comparative Work

As already mentioned, the case study presented in this paper was greatly inspired by the work of [30] 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. This will most likely require the tool support mentioned in the previous paragraph.

As already mentioned, [31] is also a good source of comparative work: It presents an Alloy model contemplating wear leveling, erase unit reclamation, and tolerance to power loss as prescribed by the ONFi industry-wide standard for the specification of NAND FLASH memory. Carrying out AoP proofs for the various assertions available from the code will be an interesting test of Alloy-meets-AoP endurance in face of a model which is more detailed than ours.

Appendix A

A.1 Alloy Auxiliary Functions

Alloy function implementing combinator $s \lhd r$:

fun ifdef[s,r : **univ** → **univ**] : **univ** → **univ** { r :> (s.dom) }

Alloy function implementing the shrinking combinator:

```
fun shrinkby[r: Key → Addr, s: Addr → Addr] :
Key → Addr {
    { a : r.dom, b : a.r | all b′ : a.r | b′ in s.b }
}
```

A.2 Basic Results of Relation Algebra

Composition:

$$R \cdot (S \cdot T) = (R \cdot S) \cdot T, \tag{75}$$

$$R \cdot id = R = id \cdot R, \tag{76}$$

$$R \cdot \bot = \bot = \bot \cdot R. \tag{77}$$

Converses:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ, \tag{78}$$

$$(R^\circ)^\circ = R, \tag{79}$$

$$R \subseteq R \cdot R^\circ \cdot R. \tag{80}$$

Union simplicity:

$$R \cup S \text{ simple} \equiv R \text{ simple} \wedge S \text{ simple} \wedge R \cdot S^\circ \subseteq id. \tag{81}$$

Union injectivity:

$$R \cup S \text{ injective} \equiv R \text{ injective} \wedge S \text{ injective} \wedge R^\circ \cdot S \subseteq id. \tag{82}$$

Shunting rule for function f:

$$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f. \tag{83}$$

Shunting rule for injective M:

$$M^\circ \cdot X \subseteq S \equiv \rho M \cdot X \subseteq M \cdot S, \tag{84}$$

$$X \cdot M \subseteq S \equiv X \cdot \rho M \subseteq S \cdot M^\circ. \tag{85}$$

Domain and range (for Φ coreflexive):

$$\delta R \subseteq \Phi \equiv R \subseteq \top \cdot \Phi, \quad (86)$$

$$\rho R \subseteq \Phi \equiv R \subseteq \Phi \cdot \top. \quad (87)$$

These offer a number of properties, namely,

$$\rho R = \delta(R^\circ), \quad (88)$$

$$\top \cdot \delta R = \top \cdot R, \quad (89)$$

$$\rho R \cdot \top = R \cdot \top, \quad (90)$$

$$R \cdot \delta R = R, \quad (91)$$

$$R = (\rho R) \cdot R, \quad (92)$$

$$\delta(R \cdot S) = \delta(\delta R \cdot S), \quad (93)$$

$$\rho(R \cdot S) = \rho(R \cdot \rho S), \quad (94)$$

$$R \cdot S^\circ \subseteq \bot \equiv \delta R \cap \delta S \subseteq \bot, \quad (95)$$

$$\delta R \subseteq \delta S \equiv R \subseteq \top \cdot S. \quad (96)$$

Complemented domain:

$$\overline{\delta S} = id \cap \bot / S^\circ, \quad (97)$$

following from

$$R \cdot \overline{\delta S} = R \cap \bot / S^\circ, \quad (98)$$

where $\bot / S^\circ$ can be replaced by $\bot / \delta S$ since they are the same. Finally,

$$R \cdot \overline{\Phi \cdot \delta R} = R \cdot \overline{\Phi}, \quad (99)$$

$$\Phi_{\neg p} = \overline{\Phi_p}, \quad (100)$$

since the domain of a coreflexive is itself.

A.3 Proofs Left Pending in Main Text

Calculation of (51):

$$\begin{array}{l}
\geq_{Jrnl} \cap \geq_{Jrnl}{}^\circ \subseteq id \\
\quad \equiv \{ \ (40); \text{twice} \ \} \\
\qquad (Jrnl \cdot \geq \cdot Jrnl^\circ) \cap (Jrnl \cdot \geq^\circ \cdot Jrnl^\circ) \subseteq id \\
\quad \equiv \{ \ \text{distribution, since Jrnl is injective} \ \} \\
\qquad Jrnl \cdot (\geq \cap \geq^\circ) \cdot Jrnl^\circ \subseteq id \\
\quad \Leftarrow \{ \ \text{since Jrnl is simple} \ \} \\
\qquad Jrnl \cdot (\geq \cap \geq^\circ) \cdot Jrnl^\circ \subseteq Jrnl \cdot Jrnl^\circ \\
\quad \Leftarrow \{ \ \text{monotonicity} \ \} \\
\qquad \geq \ \cap \geq^\circ \subseteq id.
\end{array}$$

Calculation of (63):

$$\begin{array}{l}
Flash' = Flash \cup (delN) \text{is simple} \\
\equiv \{\text{definition; union simplicity (81); Flash is simple}\} \\
\quad \begin{cases} del\ N \text{ is simple} \\ (del\ N) \cdot Flash^\circ \subseteq id \end{cases} \\
\equiv \{ \ \text{assume del N simple (see below)} \ \} \\
\quad (del\ N) \cdot Flash^\circ \subseteq id \\
\Leftarrow \{ \ delN \subseteq (N \cdot key)^\circ \text{follows from (60); converses (78)} \ \} \\
\quad Flash \cdot N \cdot key \subseteq id \\
\equiv \{ \ (56) \ \} \\
\quad \bot \subseteq id \\
\equiv \{ \ \bot \text{ is below anything} \ \} \\
\quad \text{TRUE.}
\end{array}$$

Calculation of the simplicity of $del\ N$, for N injective:

$$\begin{array}{l}
del\ N \text{ simple} \\
\quad \equiv \{ \ (5) \ \} \\
\qquad (del\ N) \cdot (del\ N)^\circ \subseteq id \\
\quad \equiv \{ \ del\ N = \overline{\delta Val} \cdot (N \cdot key)^\circ; \text{converses} \ \} \\
\qquad \overline{\delta Val} \cdot key^\circ \cdot N^\circ \cdot N \cdot key \cdot \overline{\delta Val} \subseteq id \\
\quad \Leftarrow \{ \ N \text{ is injective} \ \} \\
\qquad \overline{\delta Val} \cdot key^\circ \cdot key \cdot \overline{\delta Val} \subseteq id \\
\quad \equiv \{ \ \text{above; definition} \ \} \\
\qquad key \cdot \overline{\delta \text{Val}} \text{ injective.}
\end{array}$$

The injectivity of $key \cdot \overline{\delta Val}$ tells that there should be only one way to record deleted keys in entries, that is, the DEL mark should be unique.

Calculation of (64):

$$\begin{array}{l}
\text{Jrnl}' = Jrnl \cup M \text{ is simple} \\
\quad \equiv \{ \ (81), \text{since Jrnl and M are simple} \ \} \\
\qquad Jrnl \cdot M^\circ \subseteq id.
\end{array}$$

This is granted by condition (59) ensuring that M is right-appended to $Jrnl$, since the position of every address in M is strictly larger than that of any address in $Jrnl$. In fact, the stronger condition,

$$Jrnl \cdot M^\circ \subseteq \bot, \quad (101)$$

stems from (59):

$$
\begin{array}{l}
M^\circ {\cdot} \top \cdot Jrnl \subseteq > \\
\Rightarrow \{ \text{ since } > \text{ is ireflexive } \} \\
\quad M^\circ \cdot \top \cdot Jrnl \cap id \subseteq \bot \\
\Rightarrow \{ \text{ monotonicity } \} \\
\quad Jrnl \cdot (M^\circ \cdot \top \cdot Jrnl \cap id) \cdot M^\circ \subseteq \bot \\
\equiv \{ \text{ distribution, since Jrnl is injective and } \mathrm{M}^\circ \text{ is simple } \} \\
\quad Jrnl \cdot M^\circ \cdot \top \cdot Jrnl \cdot M^\circ \cap Jrnl \cdot M^\circ \subseteq \bot \\
\Rightarrow \{ \top \text{ is above everything; transitivity } \} \\
\quad Jrnl \cdot M^\circ \cdot (Jrnl \cdot M^\circ)^\circ \cdot Jrnl \cdot M^\circ \cap Jrnl \cdot M^\circ \subseteq \bot \\
\equiv \{ \text{ (80) } \} \\
\quad Jrnl \cdot M^\circ \subseteq \bot .
\end{array}
$$

Calculation of (67):

$$
\begin{array}{l}
index\ Flash \upharpoonright (\geq_{Jrnl'}) \\
= \{ \text{ (47); (40) } \} \\
\quad index\ Flash \upharpoonright \\
\quad (\rho(index\ Flash) \cdot Jrnl' \cdot \geq \cdot Jrnl'^\circ \cdot \rho(index\ Flash)) \\
= \{ \rho(index\ Flash) = \delta Flash \text{ (37)} \} \\
\quad index\ Flash \upharpoonright (\delta Flash \cdot Jrnl' \cdot \geq \cdot Jrnl'^\circ \cdot \delta Flash) \\
= \{ \text{ (55); (56); (57) } \} \\
\quad index\ Flash \upharpoonright (\delta Flash \cdot Jrnl \cdot \geq \cdot Jrnl^\circ \cdot \delta Flash) \\
= \{ \text{ (37) again, followed by (47) } \} \\
\quad index\ Flash \upharpoonright (Jrnl \cdot \geq \cdot Jrnl^\circ) \\
= \{ \text{definition} \} \\
\quad index\ Flash \upharpoonright (\geq_{Jrnl}).
\end{array}
$$

Calculation of (70):

$$
\begin{array}{l}
\delta Flash' \\
= \{ \text{ (54); } \cup\text{-distribution; } ci_2 \text{(52) } \} \\
\quad \rho Jrnl \cup \delta(delN) \\
= \{ \delta(delN) = \delta N^\circ \text{(60); (93) } \} \\
\quad \rho Jrnl \cup \delta N^\circ \\
= \{ \text{ (88); (57) } \} \\
\quad \rho Jrnl \cup \rho M \\
= \{ \cup\text{-distribution; (55) } \} \\
\quad \rho Jrnl'.
\end{array}
$$

Calculation of (71): Injectivity of $Jrnl'$ (55), (82) requires $M^\circ \cdot Jrnl \subseteq id$. This is granted by (101):

$$
\begin{array}{l}
Jrnl \cdot M^\circ \subseteq \bot \\
\Rightarrow \{ \text{ monotonicity } \} \\
\quad M^\circ \cdot Jrnl \cdot M^\circ \cdot M \subseteq \bot \\
\equiv \{ M \text{ injective (85); (92) } \} \\
\quad M^\circ \cdot Jrnl \subseteq \bot .
\end{array}
$$

Calculation of conjecture (73): Let us start by recording that, wherever a key participates in both N and $Flash$, its address in N is greater than any other in $Flash$:

$$
N \cdot (index\ Flash \cdot \delta N)^\circ \subseteq >_{Jrnl'} . \tag{102}
$$

We reason

$$
\begin{array}{l}
N \cdot (index\ Flash \cdot \delta N)^\circ \subseteq >_{Jrnl'} \\
\equiv \{ \text{ (34); converses } \} \\
\quad N \cdot \delta N \cdot \pi_1 \cdot Flash \subseteq >_{Jrnl'} \\
\equiv \{ \text{ (91); (55); (40) } \} \\
\quad N \cdot \pi_1 \cdot Flash \subseteq (Jrnl \cup M) \cdot > \cdot (Jrnl \cup M)^\circ \\
\equiv \{ \text{ (47)and } N,\ Flash \text{ orthogonal to respectivelyJrnl, M } \} \\
\quad N \cdot \pi_1 \cdot Flash \subseteq M \cdot > \cdot Jrnl^\circ \\
\equiv \{ \text{ (92), (57) and (88); (36) and (91) } \} \\
\quad \rho M \cdot (N \cdot \pi_1 \cdot Flash) \cdot \delta Jrnl \subseteq M \cdot > \cdot Jrnl^\circ \\
\equiv \{ \text{ shunting (84, 85) } \} \\
\quad M^\circ \cdot (N \cdot \pi_1 \cdot Flash) \cdot Jrnl \subseteq > \\
\Leftarrow \{ \top \text{ above anything; transitivity } \} \\
\quad M^\circ \cdot \top \cdot Jrnl \subseteq > \\
\equiv \{ \text{ (59) } \} \\
\quad \text{TRUE.}
\end{array}
$$

We proceed to (73). To save ink, expressions $index\ Flash \cdot \delta N$, $\geq_{Jrnl'}$, and $>_{Jrnl'}$ are abbreviated to S, $\sqsupseteq$, and $\sqsupset$, respectively. In such shortened form, (102) becomes $N \cdot S^\circ \subseteq \sqsupset$, which is the same as

$$
N \subseteq \sqsupset / S^\circ , \tag{103}
$$

introducing division (12), in turn the same as

$$
S \subseteq \sqsupset^\circ / N^\circ , \tag{104}
$$

taking converses. We reason

$$
\begin{array}{l}
(N \cup S) \upharpoonright \sqsupseteq \; = N \upharpoonright \sqsupseteq \\
\equiv \{ \text{ (49) } \} \\
\quad (N \upharpoonright \sqsupseteq) \cap (\sqsupseteq / S^\circ) \cup (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) = N \upharpoonright \sqsupseteq \\
\equiv \{ N \subseteq \sqsupset / S^\circ \text{thanks to (103) and } \sqsupset \subseteq \sqsupseteq \} \\
\quad (N \upharpoonright \sqsupseteq) \cup (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) = N \upharpoonright \sqsupseteq \\
\equiv \{ \text{ since } (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) = \bot \text{, see (105) below } \} \\
\quad N \upharpoonright \sqsupseteq \; = N \upharpoonright \sqsupseteq .
\end{array}
$$

We are left with

$$
(S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) = \bot , \tag{105}
$$

the calculation of which is as follows:

$$
\begin{aligned}
& (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) = \bot \\
\equiv & \{ \ (104) \ \} \\
& (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq / N^\circ) \cap (\sqsupseteq^\circ / N^\circ) = \bot \\
\equiv & \{ \text{ division preserves } \cap \ \} \\
& (S \upharpoonright \sqsupseteq) \cap (\sqsupseteq \cap \sqsupseteq^\circ) / N^\circ = \bot \\
\equiv & \{ \ \sqsupseteq \cap \sqsupseteq^\circ = \bot \ \} \\
& (S \upharpoonright \sqsupseteq) \cap \bot / N^\circ = \bot \\
\equiv & \{ \text{ indirect equality [8], for all suitably typed X } \} \\
& X \subseteq (S \upharpoonright \sqsupseteq) \wedge (X \subseteq \bot / N^\circ) \equiv X \subseteq \bot \\
\equiv & \{ \ (45); (12) \ \} \\
& X \subseteq S \wedge X \cdot S^\circ \subseteq \sqsupseteq \wedge X \cdot N^\circ \subseteq \bot \equiv X \subseteq \bot \\
\equiv & \{ \ (95);\ \text{S and N are not domain-disjoint} \ \} \\
& X \subseteq S \wedge X \cdot S^\circ \subseteq \sqsupseteq \wedge X \subseteq \bot \equiv X \subseteq \bot \\
\equiv & \{ \text{ trivia } \} \\
& \text{TRUE.}
\end{aligned}
$$

ACKNOWLEDGMENTS

This research was carried out in the context of the MONDRIAN Project funded by the Portuguese Science and Technology Foundation (FCT) contract PTDC/EIA-CCO/108302/2008. The authors would like to thank the anonymous referees for insightful comments which significantly improved the quality of the original submission. Comments by Raymond Boute on an earlier draft of this paper are also gratefully acknowledged. José Oliveira would like to thank António Murta for renewing his interest in Alloy.

REFERENCES

[1] *Software Engineering: Report on a Conference Sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, Seventh to 11th October 1968,* P. Naur and B. Randell, eds. Scientific Affairs Division, NATO, 1969.

[2] D.L. Parnas, "Really Rethinking 'Formal Methods'," *Computer,* vol. 43, no. 1, pp. 28-34, Jan. 2010.

[3] R. Backhouse, *Mathematics of Program Construction,* Univ. of Nottingham, 2004, draft of book in preparation.

[4] E. Kreyszig, *Advanced Engineering Mathematics,* sixth ed. J. Wiley and Sons, 1988.

[5] R. Maddux, "The Origin of Relation Algebras in the Development and Axiomatization of the Calculus of Relations," *Studia Logica,* vol. 50, pp. 421-455, 1991.

[6] G. Schmidt, *Relational Mathematics,* Encyclopedia of Mathematics and Its Applications. Cambridge Univ. Press, Nov. 2010.

[7] P. Freyd and A. Ščedrov, *Categories, Allegories,* Mathematical Library, vol. 39. North-Holland, 1990.

[8] R. Bird and O. de Moor, *Algebra of Programming.* Prentice-Hall, 1997.

[9] L. Barbosa and J. Oliveira, "Transposing Partial Components—An Exercise on Coalgebraic Refinement," *Theoretical Computer Science,* vol. 365, no. 1, pp. 2-22, 2006.

[10] L. Barbosa, J. Oliveira, and A. Silva, "Calculating Invariants as Coreflexive Bisimulations," *Proc. 12th Int'l Conf. Algebraic Methodology and Software Technology,* pp. 83-99, 2008.

[11] J. Oliveira and C. Rodrigues, "Pointfree Factorization of Operation Refinement," *Proc. 14th Int'l Conf. Formal Methods,* pp. 236-251, 2006.

[12] J. Oliveira, "Transforming Data by Calculation," *Proc. Generative and Transformational Techniques in Software Eng. II,* pp. 134-195, 2008.

[13] S. Wang, L. Barbosa, and J. Oliveira, "A Relational Model for Confined Separation Logic," *Proc. IFIP/IEEE Second Int'l Symp. Theoretical Aspects of Software Eng.,* pp. 263-270, 2008.

[14] J. Oliveira, "Extended Static Checking by Calculation using the Pointfree Transform," *Proc. LerNet ALFA Summer School Conf.,* pp. 195-251, 2008.

[15] J. Oliveira, "Pointfree Foundations for (Generic) Lossless Decomposition," Technical Report TR-HASLab:3:2011, HASLab, Univ. of Minho and INESC TEC, 2011.

[16] R. Joshi and G.J. Holzmann, "A Mini Challenge: Build a Verifiable Filesystem," *Proc. Verified Software: Theories, Tools, Experiments Conf.,* pp. 49-56, 2005.

[17] M.A. Ferreira and J. Oliveira, "An Integrated Formal Methods Tool-Chain and Its Application to Verifying a File System Model," *Formal Methods: Foundations and Applications,* vol. 5902, pp. 153-169, Springer, 2009.

[18] D. Jackson, *Software Abstractions: Logic, Language, and Analysis,* revised ed. MIT Press, 2012.

[19] R. Backhouse, *Program Construction: Calculating Implementations from Specifications.* John Wiley and Sons, Inc., 2003.

[20] S. Feferman, "Tarski's Influence on Computer Science," *Logical Methods in Computer Science,* vol. 2, pp. 1-1-13, 2006.

[21] S. Givant, "The Calculus of Relations as a Foundation for Mathematics," *J. Automated Reasoning,* vol. 37, no. 4, pp. 277-322, 2006.

[22] A. Tarski and S. Givant, *A Formalization of Set Theory without Variables.* Am. Math. Soc., 1987.

[23] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM,* vol. 21, no. 8, pp. 613-639, Aug. 1978.

[24] M.F. Frias, C.L. Pombo, and M.M. Moscato, "Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications," *Proc. 13th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 587-601, 2007.

[25] N. Macedo, "Translating Alloy Specifications to the Point-Free Style," master's thesis, Univ. of Minho, 2010.

[26] C. Jones, *Systematic Software Development Using VDM,* second ed. Prentice-Hall, 1990.

[27] D. Méry, "Refinement-Based Guidelines for Algorithmic Systems," *Int'l J. Software and Informatics,* vol. 3, nos. 2/3, pp. 197-239, 2009.

[28] T. Hoare and J. Misra, "Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project," *Proc. Verified Software: Theories, Tools, Experiments Conf.,* pp. 1-18, 2005,

[29] C.B. Jones and J. Woodcock, "Editorial," *Formal Aspects of Computing,* vol. 20, no. 1, issue devoted to the Mondex grand-challenge in verified software, pp. 1-3, 2008.

[30] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif, "Abstract Specification of the UBIFS file System for Flash Memory," *Proc. Second World Congress on Formal Methods,* pp. 190-206, 2009.

[31] E. Kang and D. Jackson, "Designing and Analyzing a Flash File System with Alloy," *Int'l J. Software and Informatics,* vol. 3, no. 1, pp. 129-148, 2009.

[32] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: Combining Specification, Proof Checking, and Model Checking," *Proc. Eighth Int'l Conf. Computer Aided Verification,* pp. 411-414, July/Aug. 1996.

[33] P. Dybjer, Q. Haiyan, and M. Takeyama, "Verifying Haskell Programs by Combining Testing and Proving," *Proc. Third Int'l Conf. Quality Software,* pp. 272-279, 2003.

[34] P. Höfner and G. Struth, "On Automating the Calculus of Relations," *Proc. Fourth Int'l Joint Conf. Automated Reasoning,* pp. 50-66, 2008.

[35] P. Matos and J. Marques-Silva, "Model Checking Event-B by Encoding into Alloy," *J. Computing Research Repository,* vol. arXiv:0805.3256v2, 2008.

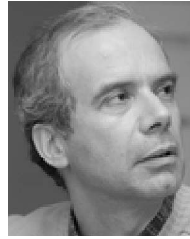
[36] C. Bolton, "Using the Alloy Analyzer to Verify Data Refinement in Z," *Electronic Notes in Theoretical Computer Science,* vol. 137, no. 2, pp. 23-44, 2005.

[37] A. Galloway, G. Lüttgen, J. Mühlberg, and R. Siminiceanu, "Model-Checking the Linux Virtual File System," *Proc. Int'l Conf. Verification, Model Checking, and Abstract Interpretation,* pp. 74-88, 2009.

[38] J.T. Mühlberg and G. Lüttgen, "Verifying Compiled File System Code," *Proc. Brazilian Symp. Formal Methods,* pp. 306-320, 2009.

[39] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," *ACM Trans. Computer Systems,* vol. 24, no. 4, pp. 393-423, 2006.

[40] A. Butterfield and A.Ó. Catháin, "Concurrent Models of Flash Memory Device Behaviour," *Proc. Brazilian Symp. Formal Methods,* pp. 70-83, 2009.
[41] A. Butterfield and J. Woodcock, "Formalising Flash Memory: First Steps," *Proc. IEEE 12th Int'l Conf. Eng. Complex Computer Systems,* pp. 251-260, 2007.
[42] K. Damchoom and M. Butler, "Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B," *Proc. Brazilian Symp. Formal Methods,* pp. 134-152, 2009.
[43] W. Hesselink and M. Lali, "Formalizing a Hierarchical File System," *Electronic Notes in Theoretical Computer Science,* vol. 259, pp. 67-85, 2009.
[44] L. Freitas, J. Woodcock, and Z. Fu, "POSIX File Store in Z/Eves: An Experiment in the Verified Software Repository," *Science of Computer Programming,* vol. 74, no. 4, pp. 238-257, 2009.
[45] C. Morgan and B. Sufrin, "Specification of the UNIX Filing System," *IEEE Trans. Software Eng.,* vol. 10, no. 2, pp. 128-142, Mar. 1984.
[46] J. Kramer, "Is Abstraction the Key to Computing?" *Comm. ACM,* vol. 50, no. 4, pp. 37-42, Apr. 2007.
[47] J.M. Wing, "Computational Thinking and Thinking about Computing," *Philosophical Trans. Royal Soc. A,* vol. 366, pp. 3717-3725, Oct. 2008.
[48] L. Russo, *The Forgotten Revolution: How Science Was Born in 300BC and Why It Had to Be Reborn.* Springer-Verlag, Sept. 2003.
[49] J. Ferreira, A. Mendes, A. Cunha, C. Baquero, P. Silva, L. Barbosa, and J. Oliveira, "Logic Training through Algorithmic Problem Solving," *Proc. Third Int'l Congress Conf. Tools for Teaching Logic,* pp. 62-69, 2011.
[50] M. in Computer Science and Engineering, "Formal Methods in Software Engineering Course," http://mei.di.uminho.pt/?q=en/mfes-uk, 30 ECTS, Univ. of Minho, 2011.
[51] J. Oliveira, "Calculating with Pointfree Alloy," contributed talk to the IFIP WG 2.1 #64 Meeting, Apr. 2009.
[52] C. Necco, J. Oliveira, and J. Visser, "Extended Static Checking by Strategic Rewriting of Pointfree Relational Expressions," Technical Report FAST:07.01, CCTC Research Centre, Univ. of Minho, 2007.
[53] P.F. Silva and J.N. Oliveira, ""Galculator": Functional Prototype of a Galois-Connection Based Proof Assistant," *Proc. 10th Int'l ACM SIGPLAN Conf. Principles and Practice of Declarative Programming,* pp. 44-55, 2008.



José N. Oliveira graduated in electrical engineering in 1978 from the University of Porto in Portugal and received the MSc and PhD degrees in computer science in 1980 and 1984, respectively, from the University of Manchester, United Kingdom. Since 2010 he has been an associate professor with habilitation at the Computer Science Department of the University of Minho, Portugal, and a member of the High Assurance Software Laboratory (HASLab) of INESC TEC/University of Minho. He is also a member of IFIP WG2.1 (Algorithmic Languages and Calculi) and of the Formal Methods Europe (FME) association.

Miguel A. Ferreira received the BSc degree in software engineering and the MSc degree in computer science from Minho University, Portugal. Currently, he is working as a researcher at Software Improvement Group (SIG). In his research at Minho University he focused mainly on software modeling for verification purposes. Later, at SIG, he has focused on different areas such as software development repository mining and subsequent empirical analysis, metrics for spreadsheets, and cloud computing. He was responsible for building the cloud infrastructure currently used by SIG to scale up its software analysis capabilities. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.