ON REGULARITY IN SOFTWARE DESIGN

R. Banach[1]

Computer Science Department, Manchester University,

Manchester, M13 9PL, U. K.

Abstract

A regular relation R, is one for which $R = R \circ R^\wedge \circ R$, where "$\circ$" is relational composition and "^" is relational transpose. By examining realistic case studies, and other examples, it is shown that when expressed using a rigorous specification notation, the majority of specifications turn out to be regular relations. This is certainly so for deterministic problems, and when abstraction relations are functions, reification preserves regularity. Nondeterministic specifications can appear to exhibit non-regularity, but at least in the most commonly occuring cases, it is argued that this is caused as much by a failure to separate concerns, as by any intrinsic lack of regularity in the specification. Such specifications can be recast into a regular form, and the process is analogous to a "transformation to orthogonal coordinates" of the original problem. A design philosophy is proposed, that places the search for regularity at the heart of specification construction, with implications for requirements capture.

1 Introduction

In this paper we set out to convince readers that certain types of relations, the regular relations, are both relevant and useful in the practice of software construction, particularly in the requirements capture and specification processes. For the sake of the precision that they can yield, we will work mainly within notations such as VDM (Jones (1990)) and Z (Spivey (1993)), though the reader will realise that the main impact of the paper is at a meta level, and thus the principal conclusions of the paper will translate to other methodologies too. Over the last couple of decades, formal specification methodologies such as VDM and Z have reached a certain maturity, and a considerable amount of experience in their use has been accumulated. When constructing a system within a framework such as VDM or Z, one generally starts with a highly abstract view of what the system is to do, and then successively refines the high level view by incorporating lower-level detail, until one is close enough to an implementation description, that one can create executable code. Each step of the refinement process introduces proof obligations

1. Email: `banach@cs.man.ac.uk`

that must be successfully discharged before one can be certain that the lower-level view accurately models the higher-level view.

Regardless of the level of abstraction, a specification of some system consists of two parts. The first is a description of the state space of the system; this essentially describes the set of configurations that an instance of the system might be in at any time. The second is a collection of specifications of operations, each of which describes how the system is to change state in response to a certain kind of stimulus, the stimulus usually coming from outside. Since a specification of an operation describes a change of state, it must involve a description of both the state before the operation commences, the pre-state, and the state after the operation has completed, the post-state. Disregarding specific details of notation, mathematically it is therefore a relation R, from the set of pre-states to the set of post-states. Although in principle this relation can be quite arbitrary, it turns out that in the vast majority of realistic cases it satisfies the property of regularity, i.e. $R = R \circ R\hat{} \circ R$.

The first aim of this paper is to establish that this is actually true. We do this mainly by referring to various examples, and to collections of case studies of specifications which predate this paper's preoccupation with regularity, and which are therefore *prima face* neutral on the issue, in particular Jones and Shaw (1990), and Hayes (1993). The second aim of the paper is to elevate the previous fact to the status of a desideratum for specifications; in other words to promulgate the view that if a specification of an operation is not regular, then perhaps there is something wrong with it. This leads to a useful discipline that can guide the earlier phases of requirements capture and specification design — "look for the regularity". This discipline is proposed at the end of this paper.

Perhaps it is as well to state plainly now what this paper does and does not set out to do. It *does* intend to bring to the fore a particular aspect (regularity) that is latent in much software design, and to promote the view that attention deserves to be paid to regularity, not only because of the structural simplicity and robustness of regular specifications, but also because of the analogies that hold between regular strategies of problem solving in the discrete world of computations, and much older strategies that apply in the world of engineering mathematics (of which more below). It *does not* intend to describe a specific software development methodology. The conclusions of the paper may be taken on board by many existing development methodologies, often in a number of different ways, depending on taste. Problem solving is (at least in conventional engineering practice) acknowledged to be a creative activity, and there is often a fertile interplay between the creative aspects and the rigorous techniques that validate them. We do not wish to be prescriptive here about how this interaction is to be managed in the case of regularity. We leave that task to the designers of specific development methodologies. As we said above, our major conclusions are intended for the meta level.

The structure of the rest of the paper is as follows. In section 2 we review relations and regular relations in particular. The criteria for regularity are many and varied, perhaps the most useful one being rationality, i.e. a relation is regular iff it can be written as $R = f \circ g\hat{}$ with f and g being partial functions. The special nature of the regularity property is probably best displayed categorically: a regular relation corresponds exactly to a bicartesian square in $\mathcal{Set}$ — this gives regular relations powerful universal properties. In section 3 we apply regular relations to deterministic specifications, and to reifications in which the abstraction "function" is indeed a (partial) function. An easy general theorem shows that these situations are regular without further ado. We quote a few examples to drive the point home. In section 4 we briefly mention that inverse deterministic specifications are regular for reasons that are analogues of the reasons in the deterministic case. In section 5 we tackle our first nondeterministic specification, one that turns out to be non-regular; until that is one changes one's perspective, whereupon regularity emerges easily. We argue that the change in perspective, a change of coordinate system to "orthonormal coordinates", is just a classical problem solving technique from engineering mathematics, which illuminates the significance of regularity from a perspective of impeccable pedigree. In section 6 we tackle the "canonical nondeterministic example", of which most nondeterministic specifications arising in practice are an instance. We argue that the non-regularity of the canonical example is attributable more to a subtle mixing of concerns than to any intrinsic non-regularity in the problem, and furthermore, that in certain cases at least, there is a failure at the requirements level which is reflected in the structure of the specification. Rewriting the specifications so as to separate concerns, or to fix the inadequacy in the requirements, leads to regularity once more. Section 7 discusses further examples, including ones which are genuinely non-regular, and comments on these. Section 8 gathers the supporting evidence from the preceding sections in order to propose a design philosophy that encourages the active search for regularity during the requirements capture and specification phases of system construction. Section 9 concludes.

2 REGULAR RELATIONS

We briefly review some material on relations in order to establish enough notation for the remainder of the paper. Good references on the properties of relations are Tarski (1941) and Suppes (1960); and much of this material is reviewed in Mili (1990), who uses it as a foundation for the study of program fault tolerence. Relations are also described in many introductory texts on discrete mathematics eg. Ross and Wright (1992). See also Schmidt and Strohlein (1993).

Let R be a relation from A to B. We write dom(R) for the domain of R, and cod(R) for the codomain of R. We write $a.R$ for the set $\{b \in B \mid aRb\}$, and $X.R$ for $\cup_{a\in X} a.R$. Likewise $R.b$ is $\{a \in A \mid aRb\}$ and $R.Y$ is $\cup_{b\in Y} R.b$. If for all $a \in A$, $a.R$ is (at most) a singleton, then we say that R is a (partial) function. Similarly if for all $b \in B$, $R.b$ is (at most) a singleton, then we say that R is an inverse (partial) function. We write R^ for the transpose, or inverse of a relation R, and $R \circ S$ for the composition of relations R and S (which works from left to right).

Now for the main definitions.

A relation R is regular iff $R = R \circ R\char`^ \circ R$. (In fact this amounts to $R \circ R\char`^ \circ R \subseteq R$ since the opposite inclusion holds for any relation).

A relation R is uniform iff $a.R \cap a'.R \neq \emptyset \Rightarrow a.R = a'.R$.

A relation R is rational iff there are (partial) functions $f : A \rightarrow P$, $g : B \rightarrow P$ such that $R = f \circ g\char`^$.

Theorem 2.1 A relation R is regular iff it is uniform iff it is rational.

The proof of the equivalence of the above criteria for regularity (and their equivalences to yet more criteria expressed in set theoretic terms) are easy enough, and can be found in Banach (1994), building on work of Mili (1990) and Jaoua et al. (1991).

Corollary 2.2 If R is regular then R^ is regular.

Corollary 2.3 If R is regular and f' and g' are suitable (partial) functions, then $R' = f' \circ R \circ g'$^ is regular.

Corollary 2.4 R is regular iff there is a bijection $\approx$ between equipollent partitions of dom(R) and cod(R) given by $aRb \Leftrightarrow [a] \approx [b]$.

All of the above follow most easily when R is expressed in rational form as $R = f \circ g$^ for some partial functions f and g. Fig. 1 below illustrates a regular relation in rational form.

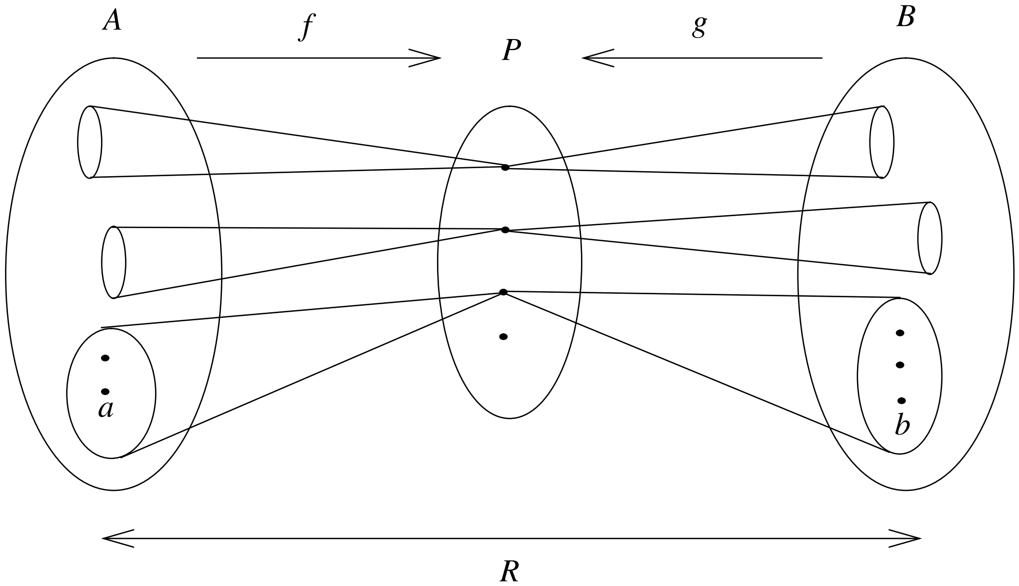

Fig. 1.

When specifications are described using regular relations, it turns out that the intermediate sets P of the rational formulations, frequently express important abstract properties of the operations. See the examples in the sections below, particularly section 6.

Universal properties

Perhaps the most striking property of regular relations is the fact that they correspond to pullbacks and bicartesian squares in $\mathcal{S}et$, the category of sets and total functions. We indicate briefly how this happens. Readers unfamiliar with category theory may simply skip the rest of this section.

Let $f : A \to P$ and $g : B \to P$ be two total functions. Their pullback is given up to isomorphism by the set K and obvious projection functions $s : K \to A$, $t : K \to B$ where

$$K = \cup_{p \in \mathrm{cod}(f) \cap \mathrm{cod}(g)} \; f^{-1}(p) \times g^{-1}(p)$$

It is clear that the elements of cod(f) $\cap$ cod(g) correspond to a bijection between blocks of partitions of $f^{-1}(\mathrm{cod}(g))$ and $g^{-1}(\mathrm{cod}(f))$ and that the elements of K thus correspond to a regular relation $R = f \circ g^\wedge$ given by $aRb \Leftrightarrow (a, b) \in K$. The fact that in the above f and g are total whereas in an arbitrary regular relation $R_0 = f_0 \circ g_0{}^\wedge$, f_0 and g_0 need only be partial, may be circumvented by enlarging P if necessary to include elements p_a and p_b, disjoint from $\mathrm{cod}(f_0) \cap \mathrm{cod}(g_0)$, and extending f_0 to all of A by sending $A - \mathrm{dom}(f_0)$ to p_a and $B - \mathrm{dom}(g_0)$ to p_b. Of course this is not the only way of making sure that f_0 and g_0 extend to total functions f and g with the same $\mathrm{cod}(f) \cap \mathrm{cod}(g)$ and same pullback object K, and so a given regular relation corresponds to many different pullback squares in $\mathcal{S}et$, even up to isomorphism.

Note that the pullback K, depends only on the restriction of f_0 to $f_0{}^{-1}(\mathrm{cod}\,(g_0))$, and the restriction of g_0 to $g_0{}^{-1}(\mathrm{cod}(f_0))$. We can use this freedom to stipulate that $f : A \to P$ and $g : B \to P$ is actually the pushout of $s : K \to A$ and $t : K \to B$. Up to isomorphism we find

$$P = [A - f^{-1}(g(B))] \uplus [\mathrm{cod}(f) \cap \mathrm{cod}(g)] \uplus [B - g^{-1}(f(A))]$$

with f and g injective on the first and last summands of P respectively. Now, both K and P are unique up to isomorphism and we have a bicartesian square in $\mathcal{S}et$, whose universal factorisation properties are illustrated in the Fig. 2 below.

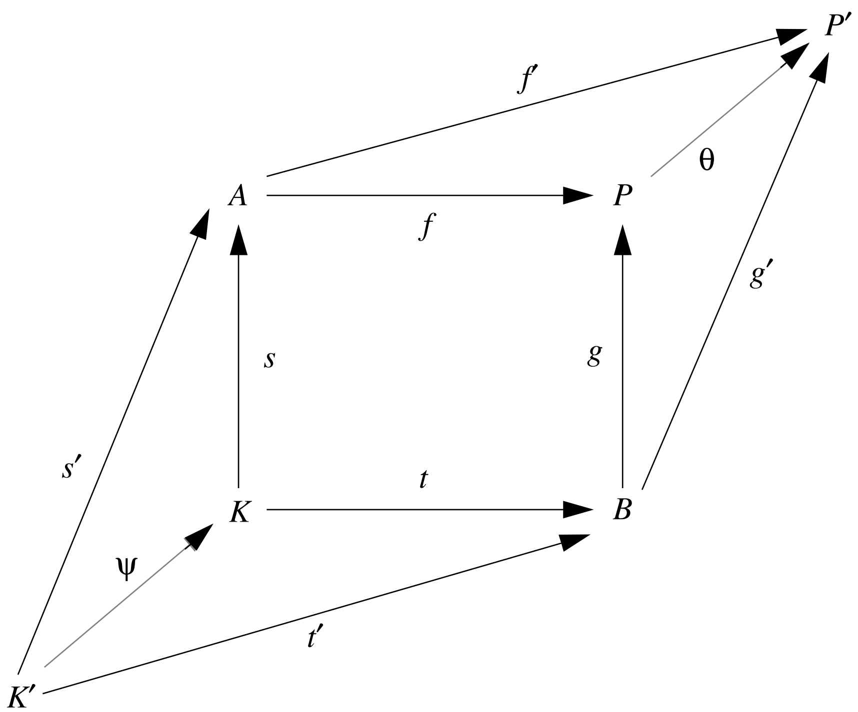

Fig. 2.

Theorem 2.5 To every regular relation R there corresponds at least one pullback square in $\mathcal{S}et$ such that the pullback object K is isomorphic to R. Also, at least one of these pullbacks is a bicartesian square, unique up to isomorphism. Conversely, to every pullback square or bicartesian square in $\mathcal{S}et$, there is a corresponding regular relation.

See Banach (1994) for a more thorough discussion and proof.

Although we do not make much use of the categorical properties of regular relations below, we mention them here for two reasons. Firstly to highlight their strong universal properties, which for a mathematical construction, is always a sign that something "special" is at issue. Secondly, it facilitates the analogy we bring out, between our methods and classical engineering-mathematical methods, as we discuss in section 5. Given that these two areas are so different, the abstractness of category theory provides an appropriately neutral playing field on which to draw out the analogy. Having thus introduced the special nature of regular relations, it should not be surprising that they display particularly convenient behaviour in applications.

3 DETERMINISTIC SPECIFICATIONS AND REIFICATION

Specifications are principally concerned with the definition of state spaces and of operations on, or between state spaces. We will take the slightly more general I/O view of operations, as translators of "inputs" from one state space A, into "outputs" in another state space B. This slightly unconventional view, which we call the "I/O picture", has been studied in eg. Hayes and Sanders (1993). The conventional position, with operations acting on a single state space, is just the special case $A = B$.

State spaces in computing applications often display a degree of structural complexity, and are usually defined by giving a suitable cartesian product of basic or already defined types, and then imposing invariants on this set to give the actual state space required. (Note that we use words such as "state space", "set", and "type" fairly interchangably in this paper, the more subtle connotations of these concepts will not be needed.)

Operations are likewise defined in pieces. Suppose we have an input state space A and output state space B already defined. An operation Π from A to B will be given by a Boolean expression on $A \times B$. If one examines the structure of this expression, one normally finds that it is a conjunction of a number of pieces. The first piece is an expression independent of B; we call it the prerestriction, $\alpha(\Pi)$; it helps to define the domain of applicability of Π. The second piece is an expression involving both A and B; we call it the trans-restriction (transition restriction), $\rho(\Pi)$; it says what the operation does. The third piece is an expression independent of A; we call it the postrestriction $\beta(\Pi)$; it helps to define the codomain of the operation Π. In a realistic specification, one or more of these pieces may be absent. At any rate, in a specification of an operation, we have a set $A \times B$ and a Boolean expression on $A \times B$ whose characteristic function defines a subset of $A \times B$, in other words a relation R from A to B.

In VDM, what we have called the prerestriction is called the precondition, and is singled out syntactically; and what for us is $\rho(\Pi) \wedge \beta(\Pi)$ is called the postcondition, also captured using special syntax. In Z, all three pieces occur together in the body of the specification, though notions of pre- and post- conditions arise in the metalanguage of Z. We have deliberately chosen a different nomenclature in order to avoid bias, and more importantly to be symmetrical between the input and output aspects of a specification.

It is important to emphasise that there is often some lattitude in how a specification is drawn up. Whether a particular criterion is captured in the structure of the basic product space of the input state, whether it is expressed in the invariant on that product space, whether it is expressed in the prerestriction α, or whether is occurs in the trans-restriction ρ, are sometimes matters of taste; and depend on such things as human readability predisposition and convenience, the expressivity of the formal notations used, the ease of manipulation of the formal notations in subsequent development stages, and so on. Similar remarks apply to the output side. And while these matters may be important in specific methodologies based on languages like VDM or Z, we want to emphasise that for us they will be tangential; we are mainly interested in hijacking the notation for its inbuilt precision when convenient. For us, all that matters is that we have a relation R from A to B, however expressed.

Definition 3.1 A specification of an operation Π, given by a relation R from A to B is deterministic iff R is a (partial) function.

Theorem 3.2 A deterministic specification R is a regular relation.

Proof. If R from A to B is deterministic then R is a partial function and $R = R \circ I_B^{\wedge}$, where I_B is the identity function on B; whence R is regular by Theorem 2.1. ☺

Union-Find

We examine a small example, the ***union*** operation from the familiar union-find problem. Let X be a fixed set. The input space consists of triples, each triple comprising a partition of X and two members of X; and the output space consists of partitions of X. In a purely relational formulation, the ***union*** operation is given by the following relation:

$$\begin{aligned} \textbf{\textit{union}} = \{\ (\langle S, p_1, p_2\rangle, S') \mid\ & S \text{ is a partition of } X \wedge \\ & \exists\, s_1 \in S \bullet p_1 \in s_1 \in S \wedge \\ & \exists\, s_2 \in S \bullet p_2 \in s_2 \in S \wedge \\ & S' = (S - \{s_1, s_2\}) \cup \{s_1 \cup s_2\}\ \} \end{aligned}$$

Since S is a partition, in the above p_1 determines s_1 uniquely and likewise for p_2, and this makes S' depend functionally on $\langle S, p_1, p_2\rangle$. At risk of labouring the point, we can write out the rational version of this specification. We have ***union*** $\subseteq A \times B$ where A is the set of triples $\langle S, p_1, p_2\rangle$ with S a partition of X, and $p_1, p_2 \in X$, and B is the set of partitions of X. We can now write:

$$\begin{aligned} &\textbf{\textit{union}} = f \circ g^{\wedge} \text{ where } f : A \to P,\ g : B \to P \text{ with } P = B,\ g = I_B \text{ and} \\ &\qquad f(\langle S, p_1, p_2\rangle) = (S - \{s_1, s_2\}) \cup \{s_1 \cup s_2\} \text{ where } p_1 \in s_1 \text{ and } p_2 \in s_2. \end{aligned}$$

Reification

Reification is an important activity in system development as it helps to bridge the gap between an abstract view of a system, and a more concrete view. In reification one starts with an abstract view of the state spaces A_o and B_o, and seeks to model them using more concrete state spaces A_c and B_c, having characteristics closer to what is regarded as being directly implementable. In the overwhelming majority of cases, the relationships between abstract and concrete state spaces are given by abstraction functions (or retrieve functions):

$\mathrm{Abs}_A : A_c \to A_o$
$\mathrm{Abs}_B : B_c \to B_o$

which are normally required to satisfy totality and surjectivity criteria:

(TOT) $\forall\, x \in A_c \bullet \exists\, y \in A_o \bullet \mathrm{Abs}_A(x) = y$
$\forall\, x \in B_c \bullet \exists\, y \in B_o \bullet \mathrm{Abs}_B(x) = y$

(SUR) $\forall\, y \in A_o \bullet \exists\, x \in A_c \bullet \mathrm{Abs}_A(x) = y$
$\forall\, y \in B_o \bullet \exists\, x \in B_c \bullet \mathrm{Abs}_B(x) = y$

(If the abstraction functions do not satisfy these criteria, then a great deal of care has to be taken in the reification process, to ensure well definedness.)

Having reified the state space descriptions, operations may be reified by composition with the abstraction functions as follows. Let Π_o be an abstract specification of an operation, and let it be given by a relation R_o from A_o to B_o. Let $\mathrm{Abs}_A : A_c \to A_o$ and $\mathrm{Abs}_B : B_c \to B_o$ be the relevant abstraction functions. Then the reified specification Π_c is given by a relation $R_c = \mathrm{Abs}_A \circ R_o \circ \mathrm{Abs}_B^{\wedge}$.

Theorem 3.3 Let $\mathrm{Abs}_A : A_c \to A_o$ and $\mathrm{Abs}_B : B_c \to B_o$ be abstraction functions. If R_o is a regular relation from A_o to B_o which describes an abstract specification Π_o of some operation, then the reified specification Π_c is given by a relation $R_c = \mathrm{Abs}_A \circ R_o \circ \mathrm{Abs}_B^{\wedge}$ from A_c to B_c which is regular.

Corollary 3.4 Let $\mathrm{Abs}_A : A_c \to A_o$ and $\mathrm{Abs}_B : B_c \to B_o$ be abstraction functions. If $R_o = f$ is a deterministic relation from A_o to B_o which describes an abstract specification Π_o of some operation, then the reified specification Π_c is given by a regular relation $R_c = \mathrm{Abs}_A \circ f \circ \mathrm{Abs}_B^{\wedge}$ from A_c to B_c.

The proofs of both of these theorems are trivial consequences of Corollary 2.3. ☺

Note that provided any resulting constraints on the domain and codomain of R_c are acceptable, Theorems 3.3 and 3.4 do not even require the totality or surjectivity criteria (TOT), (SUR) to hold.

Note further that in practice, many reifications of deterministic operations are actually deterministic subrelations of the reified specifications mentioned in Theorem 3.3 and Corollary 3.4. In such cases, the reification obviously preserves regularity also.

It is time to mention some more examples. Specifications for the operations in the simple standard textbook abstract data types will be deterministic. Things like lists, stacks, binary trees, hash tables, queues, deques, priority queues etc., are described in many places, and it is clear that the operations involved generally have a single possible output for any given input, hence are deterministic. In Appendix A we mention some more deterministic examples culled from the case study collections Jones and Shaw (1990), and Hayes (1993).

4 INVERSE DETERMINISTIC SPECIFICATIONS

By rationality, for any relation R, R is regular iff R^ is regular. Consequently, if a general specification design scheme always leads to regular relations, so will the transpose of that scheme. Applying this idea to deterministic specifications immediately yields that inverse deterministic specifications are regular. When one subsequently reifies such specifications using abstraction functions, one obviously retains the regularity property, just as for deterministic specifications.

We note a couple of examples of inverse deterministic specifications, both inspired by Hayes and Jones (1989). Both are in fact also deterministic, whence the relation between inputs and outputs is bijective. One might question the inverse deterministic formulation of the problem in such cases. However the inverse deterministic formulation may well possess a degree of conceptual clarity that is absent in a direct formulation. This may swing matters in favour of the inverse deterministic formulation.

Integer Square Root

If r is intended to be the largest integer square root of n, then we can specify the problem by writing

$$r^2 \leq n < (r + 1)^2$$

which is inverse deterministic.

Parsing

Another example arises in parsing, where one can succinctly specify the problem by the clause:

fringe(*parse_tree*) = *input_string*

where *fringe* is the usual fringe function that lists the leaves of a tree in left to right order. Again this is inverse deterministic for *parse_tree*. Generally, other clauses in the specification will narrow down the nondeterminism in the specification to the point where a given *input_string* will yield exactly one *parse_tree* (unless the grammar in question is actually ambiguous).

5 A SIMPLE NONDETERMINISTIC EXAMPLE

Now we turn to nondeterministic specifications. Our first example is relatively trivial but has lessons for us in terms of the significance of regularity. Let the input be x and the output be y both reals. The trans-restriction is:

$$\boldsymbol{lt} = \{(x, y) \mid x < y\}$$

which makes the set of related (x, y) pairs, the above-diagonal half plane. Rather obviously, 3 $\boldsymbol{lt}$ 4, 4 $\boldsymbol{lt}$^ 1, 1 $\boldsymbol{lt}$ 2; but were $\boldsymbol{lt}$ regular, we would have 3 $\boldsymbol{lt}$ 2, manifestly nonsense. So as given, $\boldsymbol{lt}$ is not regular. However let us change to rotated coordinates

$$u = y + x$$
$$v = y - x$$

then $\boldsymbol{lt}$ becomes the transformed specification $\boldsymbol{lt}^{\mathrm{r}}$ on the real variables u, v where u is unrestricted (i.e. dom($\boldsymbol{lt}^{\mathrm{r}}$) is the reals), but v is restricted to be positive (i.e. cod($\boldsymbol{lt}^{\mathrm{r}}$) is the positive reals):

$$\boldsymbol{lt}^{\mathrm{r}} = \{(u, v) \mid v > 0\}$$

Now $\boldsymbol{lt}^{\mathrm{r}}$ is regular because $u\ \boldsymbol{lt}^{\mathrm{r}}\ v$ is independent of u, whence $u_1\ \boldsymbol{lt}^{\mathrm{r}}\ v_1$, $v_1\ \boldsymbol{lt}^{\mathrm{r}\wedge}\ u_2$, $u_2\ \boldsymbol{lt}^{\mathrm{r}}\ v_2$ implies $u_1\ \boldsymbol{lt}^{\mathrm{r}}\ v_2$. By changing coordinates, we have done two things. We have recast the original problem into a form that more clearly reveals its underlying structure, and we have also subtly altered the significance of regularity with regard to the problem. This is most clearly seen when we refer to corollary 2.4. In the original formulation, we could not find suitable partitions of the x and y coordinates that related to one another in the required simple fashion. But changing to u and v allowed us to relate all the allowed values of u to all the allowed values of v.

The transformation of a problem from one set of coordinates to a more convenient set, is old hat in engineering mathematics (of which our problem may be seen as a rather trivial example). In fact one can justifiably say that a major part of classical (not to mention quantum) mathematical physics reduces to the design of appropriate coordinate systems in which the structure of problems becomes tractable. Tractability in these cases amounts to the ability to separate variables in the equations of interest. The latter, to put it in terms familiar in programming theory, is a form of divide and conquer strategy appropriate to continuous problems.

We therefore see that our search for regularity has a noble pedigree among tried and trusted problem solving techniques. Regularity corresponds to a certain separation of concerns in the problem at hand, whereby points in the problem domain which do not have much to do with one another are not brought into too close proximity as a result of using an inappropriate frame of reference to describe the problem. Often this is much easier to achieve in a continuous problem domain than in a discrete one; in the latter it is much easier to "fiddle" with arbitrary parts of the problem to destroy any uniformity of structure that may exist, and consequently there is less temptation to search for the kind of uniformity that we have been speaking about. Given the different approaches used in continuous and discrete problems, it is not too surprising that we need a fairly high level of abstraction to bring out the analogies that may exist. That separation of variables and discrete techniques have anything in common at all is interesting enough, but that their relationship might reside in something as abstract as bicartesian squares in $\mathcal{Set}$ should be less surprising.

6 NONDETERMINISTIC SPECIFICATIONS: THE CANONICAL EXAMPLE

Now we turn to a more realistic example. Operations which are inherently nondeterministic, arise when the system has some element of freedom in deciding the outcome of the operation. In the overwhelming majority of realistic cases, the choice arises because the system as a whole is growing in size, and the new data representing a quantum of growth has to be found a place in the representation of the system state. The exact place within the system state is usually of little importance, and because external users do not care about the precise details, it is left to the system to choose a place. This gives rise to the nondeterminism. We examine a typical example to see how we fare *vis à vis* regularity. The structure of this example is so common, that we call it our canonical example.

Page Allocation in a Heap

The archetypal resource allocation/deallocation scenario is dynamic storage management. Suppose we have a store containing four pages. We represent these pages by four small circles, open when free, and filled in when allocated. Suppose we are in the state [● ● ○ ○] and we receive a request for a page. We do not care which page gets allocated, so if *R* is the relation representing the allocation operation, we have

[● ● ○ ○] *R* [● ● ● ○] and [● ● ○ ○] *R* [● ● ○ ●]

depending on which free page gets allocated. On the other hand, if we are in the state [● ○ ● ○] and we request a page, we find

[● ○ ● ○] *R* [● ● ● ○] and [● ○ ● ○] *R* [● ○ ● ●]

Clearly [● ● ○ ○].*R* ∩ [● ○ ● ○].*R* = { [● ● ● ○] } ≠ ∅, but [● ● ○ ○].*R* ≠ [● ○ ● ○].*R* so that *R* is not uniform and therefore not regular by Theorem 2.1.

Operations which exhibit this behaviour invariably have two key clauses in their specifications. In the prerestriction one finds the first of them:

$$u \notin \mathrm{dom}(\overleftarrow{inuse}) \qquad (✻)$$

where *u* is a unit of resource which is about to be allocated, and *inuse* is a partial function which maps each in-use unit of resource to the data that is assigned to it, the VDM hook indicating that we are refering to the input version of *inuse*. In the trans-restriction one finds the other key clause

$$inuse = \overleftarrow{inuse} \cup \{u \mapsto datum\} \qquad (✻✻)$$

where *datum* is the thing whose use requires the allocation of *u*. (Note that we have quoted verbatim from examples with this structure. In particular we should note that *u* is implicitly a member of the type of which dom($\overleftarrow{inuse}$) is a subset, and that the *u*'s in both (✻) and (✻✻) are the same, i.e. both occur in the same scope (regardless of whether this strictly conforms to the methodology at hand).)

This nondeterministic metaphor *R* = (✻) ∧ (✻✻) displays an important asymmetry between input and output, to which the non-regularity of the specification is attributable. The clause (✻) is indifferent to which *u* outside of dom($\overleftarrow{inuse}$) we choose; all we care about is that there is such a *u*. In other words it cares not about the map $\overleftarrow{inuse}$ itself, but only about the multiset of values

$$\overleftarrow{Val} = \{datum \mapsto n \mid datum \in \mathrm{cod}(\overleftarrow{inuse}) \wedge n = |\{x \in \mathrm{dom}(\overleftarrow{inuse}) \mid \overleftarrow{inuse}(x) = datum\}|\ \}$$

that dom($\overleftarrow{inuse}$) refers to[1]. On the other hand, the clause (✻✻) is fussy about the map *inuse* itself, since it demands that $\overleftarrow{inuse}$ and *inuse* differ only on u.

This is a significant mixing of levels of abstraction. Clause (✻) is abstract in that (implicitly) it only demands that enough dom($\overleftarrow{inuse}$) objects are available to refer to the multiset $\overleftarrow{Val}$; there are many different $\overleftarrow{inuse}$ maps that will do the job. However (✻✻) is concrete in that it is specific about the map *inuse* itself in demanding that it changes as little as possible. The latter is an efficiency consideration — obviously it is ludicrous to remap already allocated values whenever a new datum requires allocation, moreover since members of dom(*inuse*) are frequently passed around other parts of the system, as nicknames for the data that *inuse* maps them to; genuinely remapping the already mapped data would obviously incur substantial overheads. (Nevertheless we point out that copying garbage collectors, when copying the live data into the unused halfspace in response to an allocation request which triggers a collection operation, come close to exactly this behaviour.) So we have a mixing of concerns: an abstract view implicit in the prerestriction, and a more concrete view in the trans-restriction. In view of our remarks about separation of concerns in section 5, it is not surprising that the specification is poorly behaved *vis à vis* regularity.

What happens when we try to unmix the concerns? At the abstract level we should care only that the *inuse* maps cater for the appropriate multisets of values, so in the prerestriction we will find

$$\exists\, u \in U \bullet u \notin \mathrm{dom}(\overleftarrow{inuse}) \qquad (✻_0)$$

(where U is the correct type for u). Of course this is no different from (✻) above except that we are being more precise about the scope of u. In the trans-restriction we will then have

$$Val^{-1}(datum) = \overleftarrow{Val}^{-1}(datum) + 1 \qquad (✻✻_0)$$

which says that *Val* contains one extra instance of *datum* compared with $\overleftarrow{Val}$, and which implicitly specifies *inuse* nondeterministically through the formula for *Val*. We claim that $R_0 = (✻_0) \wedge (✻✻_0)$ gives rise to a regular relation, since any $\overleftarrow{inuse}$ that yields $\overleftarrow{Val}$ will be related to any *inuse* that yields *Val*. Let us substantiate this by displaying R_0 in rational form. Let D be an appropriate type for *datum*. Then the type of *inuse* maps is $U \to D$. The type of R_0 becomes $R_0 \subseteq (U \to D, D) \times (U \to D)$. Writing $\mathcal{M}T$ for the type of multisets over T and $\uplus$ to insert an element into a multiset we can write

$$R_0 = f \circ g^\wedge \text{ where}$$
$$f : (U \to D, D) \to \mathcal{M}D : (\overleftarrow{inuse}, datum) \to \overleftarrow{Val} \uplus datum$$
$$g : (U \to D) \to \mathcal{M}D : inuse \to Val$$
$$\text{where } Val = \overleftarrow{Val} \uplus datum$$

---

1. Obviously, the codomain of a map is in reality a set, not a multiset. We use a multiset rather than a set in this discussion in order to keep track of the multiplicity of elements of dom(*inuse*) that refer to a given element of the codomain; this is significant at this level of abstraction.

It is now clear that R_o is regular. Of course the view expressed by R_o is indifferent to the costs of remapping mentioned above.

Latent in the above is a yet more abstract view of the allocation problem. Factoring the map f above as

$f = f_A \circ f_{oo}$ where

$$f_A : (U \to D, D) \to (\mathcal{M}D, D) : (\overleftarrow{inuse}, datum) \to (\overleftarrow{Val}, datum) \text{ and}$$

$$f_{oo} : (\mathcal{M}D, D) \to \mathcal{M}D : (\overleftarrow{Val}, datum) \to \overleftarrow{Val} \uplus datum$$

we find that the essence of the problem may be understood using $R_{oo} = f_{oo}$ alone. R_{oo} expresses the notion that what is of interest is soley the data. This view is appropriate at the most abstract levels of system description, those in which representation issues are of no interest. If we disregard the fact that *Val* implicitly encodes the *inuse* maps, we can write this highest level specification, Z-style for variety, as

$$\begin{array}{|l}
\hline
\quad R_{oo} \\
\hline
datum? \quad : D \\
Val? \quad : \mathcal{M}D \\
Val! \quad : \mathcal{M}D \\
\hline
Val! = Val? \uplus datum? \\
\hline
\end{array}$$

R_o may now be obtained as a reification of R_{oo} via

$$R_o = f_A \circ f_{oo} \circ g^\wedge$$

Depending on our point of view we can regard R_{oo} in different ways. On the one hand, we can see it as having revealed a useful clarification of the problem by bringing out extra abstraction as a byproduct of our search for a regular description of the original situation. (Note by the way, that R_{oo} is deterministic, it is just the relation that relates multisets which differ by one member — thus we might be justified in regarding the whole problem as belonging to the deterministic camp of section 3.) On the other hand, we could say that that R_{oo} loses too much of the detail that we are regarding as essential at the current level of abstraction, and thus that R_{oo} is mainly a manifestation of alternative ways of viewing the intermediate set P in the rational representation of R_o as $f \circ g^\wedge$ with $f : A \to P$ and $g : B \to P$.

Amongst other things, this last remark highlights the usefulness in general of searching for an intermediate set P in the rational description of a problem that is "as informative as possible". Theorem 2.1 assures us that for a regular relation a suitable P always exists, but the canonical construction of P in the proof is not enlightening. However, a proper understanding of the problem can often show that the canonical set is isomorphic (in the category $\mathcal{S}et$), to some set which captures some characteristic properties of the problem. To that extent, finding a good P for a regular problem is one of the creative aspects of problem solving. We alluded to this already in section 2. The improved understanding of the problem that a good P witnesses, can lead to benefits further down the line, such as correct designs and implementations.

We return now to R_{o} which tells only the more abstract half of the original story. To say something about efficiency, we must reify R_{o}. In any realistic situation, the storage type U would be managed by some storage manager, among whose tasks would be the selection of unused locations u when an allocation needs to be made. Normally, efficiency considerations dictate that the operation of the storage manager would be deterministic, so we can express the selection of an unused location u by the use of a deterministic choice function *best* that chooses for us the "nearest" (or in some other sense "best") as yet unallocated object in U, given that we wish the already allocated part of U to remain unaltered. Note that the operation of the storage manager would invariably depend on some internal data structures that are of no concern to us here. In fact we should regard *best* as a free name in the clauses below, unspecified apart from being restricted to refer only to deterministic choice functions on U. This is a subtly different expression of the nondeterminism in the problem compared with (✻). The prerestriction then becomes:

$$\exists\, u \in U \bullet u = best(U - \mathrm{dom}(\overleftarrow{inuse})) \qquad (✻_{\mathrm{c}})$$

and the trans-restriction becomes:

$$inuse = \overleftarrow{inuse} \cup \{u \mapsto datum\} \qquad (✻✻_{\mathrm{c}})$$
$$\text{where } u = best(U - \mathrm{dom}(\overleftarrow{inuse}))$$

It is clear that $R_{\mathrm{c}} = (✻_{\mathrm{c}}) \wedge (✻✻_{\mathrm{c}})$ is deterministic and thus regular, so that latent in our original non-regular specification we find regularity, revealed by a suitable separation of concerns, and a reification of $(✻_{\mathrm{o}}) \wedge (✻✻_{\mathrm{o}})$ to $(✻_{\mathrm{c}}) \wedge (✻✻_{\mathrm{c}})$. We summarise the relationship of the three versions of the problem that we have introduced below. We see that only the middle one is genuinely nondeterministic.

$$R_{\mathrm{c}} \underset{\text{Reification}}{\overset{\text{Abstraction}}{\rightleftarrows}} R_{\mathrm{o}} \underset{\text{Reification}}{\overset{\text{Abstraction}}{\rightleftarrows}} R_{\mathrm{oo}}$$

The metaphor (✻) ∧ (✻✻) is almost universal in the majority of realistic nondeterministic specifications, as these usually arise when an operation has to allocate an item from an internally managed resource, to contain, or represent, or refer to new data generated because the operation is one that caused the system to grow. In Appendix B we list some operations from our aforementioned collections that fit this paradigm. All may be reworked along the lines described above.

We have just covered the canonical example in some detail, but there is a further surprise in store. We remarked casually above that the (✻) ∧ (✻✻) metaphor is used when an operation (let's call it ***allocate***) allocates a slot for a new datum in some storage medium, with the chosen member of dom(*inuse*), u say, passed around as a nickname for the stored datum. But how is this member of dom(*inuse*), u, actually passed to users of ***allocate***? Well such users could compare dom(*inuse*) to $\mathrm{dom}(\overleftarrow{inuse})$ provided they had wisely retained a copy of the latter somewhere beforehand. But this is hardly in the spirit of the way operations such as ***allocate*** are intended to work. As given above, (perhaps slightly deviously, as the specifications were not written out in full), there is an implicit existential quantification of u over the body of the specification, making u in a sense private to the operation. (This last remark must be viewed with caution as *inuse* is externally visible — though formalisms such as VDM and Z do not prescribe *how far* it might be visible — nevertheless the identity of u is to a greater or lesser extent obscured.) This of course is no good to users of ***allocate***. These need u explicitly, so that u must be an output parameter. Let us make this change. Using Z notation we find

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{R_{\text{if}}} \\
\hline
inuse? \quad : U \to D \\
datum? \quad : D \\
inuse! \quad : U \to D \\
u! \quad : U \\
\hline
u! \in U \wedge \\
u! \notin \mathrm{dom}(inuse?) \wedge \\
inuse! = inuse? \cup \{u! \mapsto datum?\} \\
\hline
\end{array}$$

This innocent change has a startling effect. Noting that when we delete $\{u! \mapsto datum?\}$ from *inuse*! we get a unique *inuse*?, we see that R_{if} is in fact inverse deterministic and thus regular. A simple repackaging of the data involved (actually a change of signature of the operation to $(U \to D, D) \times (U \to D, U)$), has thus brought out the regularity right away. This is because there is no longer any ambiguity about which $u!$ has been allocated, even if *datum*? is a value that is already present elsewhere in the map *inuse*?.

What are we to learn from this? First of all, both the earlier R_{oo}, R_{o} and R_{c} are fine as specifications *in themselves*. But R_{if} captures the intuitive behaviour of an ***allocate*** operation considerably better than the others. In fact it is not unreasonable to to regard (✻) ∧ (✻✻) and R_{oo}, R_{o}, R_{c} as containing *requirements errors*, insofar as they are intended to describe the behaviour of ***allocate***, because they hide the identity of $u!$. They would be akin to bank deposit operations that did not tell the depositor which account had been credited. At the heart of this phenomenon is the issue of information loss, because of the way (✻) ∧ (✻✻) and R_{oo}, R_{o}, R_{c} treat $u!$. We will have more to say about the connection between information loss and regularity later on.

Thus we have a concrete case in which concentrating on regularity has brought to light problems at the requirements level, a possibility fully in line with the intentions of this paper. Essentially the point is this: if the mapping *inuse* is central to the description (at the current level of abstraction), it is most likely of interest to other operations (at the current level of abstraction) to know the identity of $u!$ when it is assigned; on the other hand, if other operations (at the current level of abstraction) have no need of $u!$, to what extent is the map *inuse* relevant at all (at the current level of abstraction). Readers may care to examine the operations mentioned in Appendix B in their proper context with regard to these observations.

We end this section by remarking that some specifications that contain the (✻) ∧ (✻✻) metaphor actually embody the transpose of the situation described by R_{if}, in particular being deterministic. This happens when the $u \in U$ that appears in the metaphor is part of the input to the operation (rather than the output as in R_{if}). Under these circumstances, the operation is not free to choose u at will, the choice coming from the environment. A particular example of this noncanonical use of the metaphor is the ***clocking_in*** operation from the FLEXITIME case study in Hayes (1993).

Thus in summary, there are three distinctive scope scenarios for specifications containing the ($\circledast$) $\wedge$ ($\circledast\circledast$) metaphor. The u in question may be an input, making the specification deterministic; or an output, making it inverse deterministic — in both these cases u is free in the body of the specification and the specification is manifestly regular; or it may be "hidden" or implicit, i.e. existentially quantified, bound in the body, as in R_0 say, whereupon the regularity is closely tied up with clean separation of concerns.

7 OTHER NONDETERMINISTIC EXAMPLES

In this section we examine some more nondeterministic specifications. These do not fit into the canonical example paradigm of the previous section. We look at them so as to see how they fare *vis à vis* regularity. They are ordered by the ease with which they conform to the regular paradigm, the early ones being the more obviously regular.

Short_Count in ISTAR

The ***short_count*** operation from the ISTAR case study in Jones and Shaw (1990) has the key clauses

$$(\overleftharpoon{count} \le 1 \Rightarrow result = \overleftharpoon{count}) \wedge$$
$$(\overleftharpoon{count} > 1 \Rightarrow result > \overleftharpoon{count})$$

We note that this is nondeterministic and regular since any value > 1 of $\overleftharpoon{count}$ is related to any value > 1 of *result*.

Random Numbers

A ***choose_random_number*** operation may be given in VDM-ese by

$$
\begin{array}{l}
\textbf{\textit{choose\_random\_number}}\ ; \\
\quad \begin{array}{ll}
\textsf{\textbf{wr}} & n : [0\,..\,N]\ ; \\
\textsf{\textbf{pre}} & true \\
\textsf{\textbf{post}} & true
\end{array}
\end{array}
$$

which means that the value of n is allowed to change at will within the range $0\,..\,N$. Therefore each input value of n is related to each output value of n, and the operation is regular. This formulation does of course ignore the statistical properties of sequences of invokations of the ***choose_random_number*** operation that are rather important in practice.

Statistical Desk Calculators

A statistical desk calculator furnishes an interesting example where reification takes place using an abstraction relation rather than an abstraction function. As a simple example, we examine the ***mean*** operation at both abstract and concrete levels.

At the abstract level, the calculator collects the numbers input by the user, forming the multiset *numbs*, and the ***mean*** operation just yields the mean of these numbers. Note that these are deterministic operations. Below we use $\uplus$ to add an element to a multiset, Σ to sum the elements of a multiset of numbers, and $|\ |$ for the cardinality of a multiset.

$\boldsymbol{input\_number}_{\mathrm{o}}\ (\ x : \mathrm{Int}\ )\ ;$

wr $numbs_{\mathrm{o}} : \mathcal{M}\mathrm{Int}\ ;$

pre $true$

post $numbs_{\mathrm{o}} = \overleftarrow{numbs_{\mathrm{o}}} \uplus \{x\}$

$\boldsymbol{mean}_{\mathrm{o}}\ ;$

rd $numbs_{\mathrm{o}} : \mathcal{M}\mathrm{Int}\ ;$

wr $mean_{\mathrm{o}} : \mathrm{Real}\ ;$

pre $true$

post $mean_{\mathrm{o}} = \Sigma\ numbs_{\mathrm{o}} \,/\, |\ numbs_{\mathrm{o}}\ |$

At the concrete level, a desk calculator does not maintain a multiset, which could grow arbitrarily large, but just maintains a running mean and running cardinality, which it updates as each new number arrives. When the mean is requested, its value is just output.

$\boldsymbol{input\_number}_{\mathrm{c}}\ (\ x : \mathrm{Int}\ )\ ;$

wr $card_{\mathrm{c}} : \mathrm{Int}\ ;$

$mean_{\mathrm{c}} : \mathrm{Real}\ ;$

pre $true$

post $(\ \overleftarrow{card_{\mathrm{c}}} = card_{\mathrm{c}} + 1) \wedge$

$(mean_{\mathrm{c}} = (\ \overleftarrow{card_{\mathrm{c}}} \times \overleftarrow{mean_{\mathrm{c}}} + x) \,/\, card_{\mathrm{c}})$

$\boldsymbol{mean}_{\mathrm{c}}\ ;$

rd $mean_{\mathrm{c}} : \mathrm{Int}\ ;$

pre $true$

post $true$

Clearly many different multisets of numbers will give rise to the same mean, although not to the same history of running means; so from the point of view of an individual instance of the ***mean*** operation, abstraction is a function from the abstract to the concrete view rather than the other way round. In fact this is a consequence of the finite nature of the concrete view. Transform theory teaches us that if we were to maintain an infinite number of running moments of the distribution being analysed (something even more unrealistic than maintaining knowledge of the complete multiset), then the relationship between abstract and concrete worlds would be bijective. Because we maintain only a finite number of moments in reality, we lose information. In this context it is not surprising that abstraction is not an arbitrary relation, but a function from abstract to concrete views as mentioned; nor that in both views the ***mean*** operation is deterministic, hence regular.

Float Square Root

A floating point ***square_root*** operation may be specified, following Hayes and Jones (1989), by

square_root ;
rd x : Float ;
wr *root* : Float ;
pre $x \geq 0$
post $\exists\, r : \text{Real} \bullet r^2 = x \wedge |\, root - r \,| < 0.01$

The type Float, of machine-representable reals, is introduced to deal with unavoidable machine imprecision. As a result, the answer of the operation, *root*, has to be constrained no more tightly than by the width of a given window (of width 0.01 in our case). This yields nondeterminism. As x increases through nearby values, these windows will typically overlap properly. Thus we will have $x_1.R \cap x_2.R \neq \emptyset$ but $x_1.R \neq x_2.R$, where R is the relation representing the specification, and x_1, x_2 are two nearby input values, and this leads to non-regularity. We can recover this situation using the strategy of section 5. Let us define a fresh coordinate for the right half of the (x, *root*) plane

$$w = root - \sqrt{x}$$

Then the key clause in the specification becomes

$$|\, w \,| < 0.01$$

which is now regular. This is a pleasing result, but obscures one point. The space of Floats is a discrete subspace of the Reals (although these days, it may be regarded as an adequately dense one), therefore the set of points ranged over by w will not coincide exactly with the set ranged over by *root*, since an exact square root is involved in the definition of w. However the density of the Floats is such that few would disagree that this is, in practice, a trivial difficulty.

Raster Graphics

In the LINE REPRESENTATION ON GRAPHICS DEVICES case study in Jones and Shaw (1990), the inability to represent a line in the real plane exactly on a raster device, leads to a nondeterministic specification of those sets of pixels which are acceptable approximate representations. This leads again to a window-style technique, whereby a sausage-like region of the pixel array surrounding the ideal line, constrains the approximations that are permitted. As the ideal line moves through nearby values, the saugages overlap and a specification that is at face value non-regular results.

In principle this may again be attacked by transformation of coordinates techniques. Given a line segment, erect an elliptical coordinate system such that the line segment connects the two foci of the ellipse. If we call the radial coordinate w, we may specify suitable approximations to the line segment by (say) the set of convex regions of the plane which contain the region $w \leq \varepsilon$ (for ε suitably small), and are themselves contained within the $w = \kappa$ contour, where κ is suitably large. If the construction of the elliptical coordinate system is smoothly parameterised in terms of the line segment endpoints, the set of regions that results (expressed in the elliptical coordinates) is independent of the line segment, and the problem becomes regular.

Though pleasing, this approach might be viewed with some caution. Firstly, the sausage shapes in the original case study are not exactly elliptical. This not much of a problem, since in principle one can invent orthogonal coordinate systems other than the elliptical one, with sausage shaped contours whose exact form was other than an ellipse. This would be technically arduous rather than conceptually challenging. Secondly, the output of the representation is intended to be sets of pixels rather than regions of the plane, and so we run into a difficulty similar to the definedness of *w* in the floating square root case study above: the well definedness of the elliptical coordinate system and of the concept of convex region with the stated properties does not easily translate into a definition of corresponding sets of pixels in a grid. (Simply refering to the set of pixels falling within an acceptable region is not enough since this has to be parameterised in elliptic terms.) Thirdly and most importantly, the resolution of raster devices in depicting the plane does not come close to the density of Floats as approximations to the Reals. It is unrealistic to ignore edge effects as was reasonable to do in the square root case. To achieve genuine regularity in the problem, given two different line segments, we would need to find a bijection between the sets of acceptable pixel collections that approximated the two lines, by analogy with the conformal transformation of the region between two ellipses that would do the job in the continuous case. Given the resolution of raster devices, it is unrealistic to expect this in general, and so, the degree to which the present case study fails to be regular is attributable to the uneven way in which a simple grid is able to approximate an implicitly continuous situation.

We can regard this as a consequence of loss of information. Different line segments are approximable with different degrees of accuracy, and as one moves from a line segment having a large number of acceptable approximations to one having fewer ones, there is a loss of information signalled by the lack of a bijection between them. Of course it may be the case that if one considers not lines in the real plane, but "floating" lines in the 2D Float plane, then for a suitable resolution of the Floats, this difficulty would be ameliorated, as the problem would then transform to the coarsening of one raster pattern into another. In any event the problem is rather sensitive to edge effects, and this constitutes a useful observation in itself.

As an example of a situation where the approximation works smoothly, and there is an obvious bijection between approximations at different points of the domain, we could mention the familiar *floor* and *ceiling* functions on the reals, that give two different ideas of an integer approximation to a real number. For either of these functions there is exactly one approximant for each real, and so the problem is deterministic, hence regular.

Nondeterministic Merge

A good source of nondeterminism comes from merging and permutation problems. A good example is nondeterministic merge. Suppose Alpha is a suitable alphabet.

ND_merge ;
rd X : seq of Alpha ;
Y : seq of Alpha ;
wr Z : seq of Alpha ;
pre *true*

post $Z \in \mathit{shuffles}(X, Y)$

$$
\begin{aligned}
\text{where}\quad & \mathit{shuffles}([\,], Y) = \{Y\} \\
& \mathit{shuffles}(X, [\,]) = \{X\} \\
& \mathit{shuffles}(x :: xs, y :: ys) = \\
& \qquad \{x :: X' \mid X' \in \mathit{shuffles}(xs, y :: ys)\} \cup \\
& \qquad \{y :: Y' \mid Y' \in \mathit{shuffles}(x :: xs, ys)\}
\end{aligned}
$$

At the core of nondeterministic merge is the following observation. Let $N^A = [\, 1^A, 2^A, 3^A, \dots \,]$ and $N^B = [\, 1^B, 2^B, 3^B, \dots \,]$ be two disjoint sequences of tags. If X and Y are two initial segments of N^A and N^B, then from any $Z \in$ *shuffles*(X, Y) we can uniquely reconstruct X and Y simply by looking at the tag superscripts in order. This means that the ***ND_merge*** problem with sequences from disjoint alphabets is inverse deterministic and thus regular. Many realistic applications of nondeterministic merge are of this character as the sequences of items being merged are tagged with their sequence of origin. This is particularly so in the field of operating systems, where servers of various kinds service requests from a number of sources.

The version of the problem given in the specification above, where the sequences are from the same alphabet, can be obtained from the previous case by applying suitable alphabetic morphisms $\Phi^A : N^A \to \text{Alpha}$, $\Phi^B : N^B \to \text{Alpha}$. If $\text{rng}(\Phi^A) \cap \text{rng}(\Phi^B) \neq \varnothing$ then loss of information occurs and the operation becomes non-regular. For example, if Alpha = char, then if X = “a” and Y = “ab” then *shuffles*(“a”, “ab”) = {“aab”, “aba”}; while if X = “a” and Y = “ba” then *shuffles*(“a”, “ba”) = {“aba”, “baa”}. So the images of (“a”, “ab”) and of (“a”, “ba”) under ***ND_merge*** have an element in common but do not coincide, i.e. ***ND_merge*** is not uniform. However, when this occurs, for any pair of pairs of sequences having properly intersecting *shuffle* sets, there will be a closed system of merge instances satisfying a higher order permuatability equation (see the conclusions). For instance in our given case, the system is closed by X = “b” and Y = “aa” with *shuffles*(“b”, “aa”) = {“aab”, “aba”, “baa”} and we get the equation

$$R \circ R\hat{} \circ R \circ R\hat{} = R \circ R\hat{}$$

for the relation that relates pairs of strings to members of their shuffle sets. Once again we see that loss of information can lead to a loss of regularity.

Diff

Another slightly unusual example, mentioned in Hayes and Jones (1989), is the UNIX ***diff*** utility. This accepts two files (sequences of lines of characters), as input, and as output, produces a set of edits that converts the first file into the second. Again this is highly nondeterministic, (eg. the global edit which just replaces all of the first file with all of the second will always work).

To study this example, let us examine the simpler ***sdiff*** which just edits one sequence X into another Y by outputting a set of substitutions of slices of X by slices of Y, each in the form $[n \,..\, m] \mapsto [n' \,..\, m']$, and each of which is intended to signify that the subsequence from n to m inclusive of X is to be replaced by the subsequence from n' to m' inclusive of Y. Assuming a suitable suite of invariants to ensure that edits are well defined and consistent etc., we may write the top level specification as

sdiff ;
rd X : seq of char ;
Y : seq of char ;
wr EDS : set of subst ;
pre *true*
post $ApplyEdits(EDS, X) = Y$

Let us pursue the strategy that worked in the ***ND_merge*** example. Let $N^A = [\ 1^A, 2^A, 3^A, \ldots\ ]$ and $N^B = [\ 1^B, 2^B, 3^B, \ldots\ ]$ be two disjoint sequences of tags as previously, and let X and Y be two initial segments of N^A and N^B. Then (up to unimportant variations), there is only one edit that will do the job, the one that always works, replacing all of X by all of Y. We conclude that this subproblem of the general ***sdiff*** problem is inverse deterministic and thus regular. Again we obtain the general case by applying suitable alphabetic morphisms $\Phi^A : N^A \to \mathrm{Alpha}$, $\Phi^B : N^B \to \mathrm{Alpha}$ and if $\mathrm{rng}(\Phi^A) \cap \mathrm{rng}(\Phi^B) \neq \varnothing$ then loss of information occurs and the operation becomes non-regular. We illustrate this by the letting Alpha = char, X = "ApqrBxyzC" and Y = "A12B4B56C". The two edits which work, are

$$E1 = \{[2\,..\,4] \mapsto [2\,..\,5],\ [6\,..\,8] \mapsto [7\,..\,8]\},\ \text{and}$$
$$E2 = \{[2\,..\,4] \mapsto [2\,..\,3],\ [6\,..\,8] \mapsto [5\,..\,8]\}$$

E1 also converts "ApqrBxyzC" into "A1234B56C", but E2 doesn't. So we have a non-uniform example as claimed. As before, we may perform deeper analyses of the problem, whereupon we would once again encounter higher order permutability equations for the relation describing the specification.

Recap

Let us comment on the above. We have encountered varying degrees of success in taking a range of nondeterministic examples, and showing in what sense they can be viewed as displaying regularity. The earlier examples, ***short_count***, ***choose_random_number*** and the statistical calculator example were regular *ab initio*. The ***square_root*** example could be transformed into regular form using a change of variable, and the process was reasonably convincing. The same approach yielded a strategy for the raster graphics example, but here the result was considerably less convincing due to the coarseness of the pixel grid on a realistic device. This caused a potential loss of information to take place which was much less innocuous than in the square root case, even though mathematically the underlying phenomenon was the same. This loss of information was correlated with a corresponding loss of regularity. Even sharper cases of loss of information appeared in the final two examples, ***ND_merge*** and ***sdiff***, where with enough combinatorial effort, one could attempt to quantify the loss of information through the way that the overlap in the ranges of two alphabetic morphisms identified cases which would otherwise have remained distinct. We saw there, that allied to this loss of regularity was the relevance of higher order permuatbility equations for the relations in question, though there is clearly a need for a more thoroughgoing analysis than we have given above.

8 THE REGULAR DESIGN PHILOSOPHY

Aside from some cases discussed in section 7, where we found evidence that higher order permutability was relevant, we have found on the whole, that specifications, and thus the underlying problems that they describe, can be understood using regular relations, (and, particularly in section 6, that this approach could be beneficial at the requirements level). This is certainly true of the overwhelming proportion of "industrial" case studies in the Jones and Shaw, and Hayes collections. We are therefore on solid ground in proposing that regularity is an inherent property of the specifications of "real-world" or "practical" problems; problems that people are actually likely to need to solve.

One can propose good reasons for this. In the real world, people act with specific goals in mind — usually. At any rate when a system of real-world procedures is computerised, the range of possible behaviours tends to get narrower rather than broader. Even when there is a range of acceptable outputs from a given input, these often represent a freedom of implementation choice for some more abstract single-valued goal. Regular relations have just the right properties to capture such situations. If $R = f \circ g^\wedge$ is a regular specification, with $f : A \to P$, $g : B \to P$, then $C = \mathrm{cod}(f) \cap \mathrm{cod}(g) \subseteq P$ is the set of values of the most abstract characterisations of the operations in question. For example, in our canonical ***allocate_page*** nondeterministic operation, the elements of C can be taken to correspond to the possible values taken by the *Val* multisets formed from the data mentioned in the discussion of $(\ast_{\mathrm{oo}}) \wedge (\ast\ast_{\mathrm{oo}})$ in section 6.

Working at the level of abstraction represented by the set C can often be alien to conventional thinking about the problem in hand. For instance this was true of our canonical example, which needed reformulation in order to bring out the regularity. We regard such reappraisal of a problem as entirely healthy. Not only can it lead to a better understanding of the symmetries of the situation, and thence to a cleaner reification strategy, but in many cases it can actually turn out that this most abstract view of an operation, characterised by C, is deterministic (eg. our canonical example again). Determinism is generally easier to deal with conceptually, so we regard its discovery in *a priori* nondeterministic situations as beneficial.

Regularity can give us a cleaner reification strategy because it breaks up both the domain and codomain of the specification into independent pieces, each (corresponding pair of which) can be dealt with separately. This can help to structure the extra levels of detail of reifications and to reduce the complexity of the proofs entailed by refinement, effectively by pulling out case analysis to the top level, and replacing large monolithic proofs by a collection of shorter derivations for each of the possible cases. This paper is not the only place where such a decomposition of input and output spaces is recommended. For example, the tabular methods of Parnas (1992) are also based on a decomposition of domain and codomain into independent pieces. Other related remarks on specification structure and ease of verification/validation (though not on regularity), can be found in Mili et al. (1986). The decomposition into independent pieces is very desirable given that the greater detail of a reified representation always carries with it a greater risk of clutter, chaos and error. Starting from a regular specification, it is easier to reify in a “balanced” way, adding equivalent layers of detail at the input and output sides of operations. (Of course this is less of a problem when input and output state spaces coincide.) We summarise some recommendations to this effect in the slogans that appear in Fig. 3.

The slogans of Fig. 3 take into account that there are cases where regularity does not apply. This may happen for a number of reasons ranging from the benign to the serious. It may be that all that is needed is a suitable change of coordinates, as in some of our examples above. Then the question arises whether the change of representation is worth pursuing at the implementation level. The issues that have to be weighed here include whether or not computing the transformation would itself be equivalent to solving the whole problem (as was true in our examples), and if not, whether computing the transformation is an efficient implementation strategy in its own terms. If computing the transformation is not efficient, it may still be worth pursuing because of the simpler problem structure that is revealed when the original problem is cast into regular form; giving payoffs in terms of more straightforward implementation and future maintainability. More seriously, the original problem may resist being put into regular

The Regular Design Philosophy

- ❇ When commencing the design of a system, look for regularity from the earliest possible moment. Make the search for regularity central to the requirements capture phase, as well as to the specification phase. Understand the significance of suitable sets $C = \mathrm{cod}(f) \cap \mathrm{cod}(g)$ in a rational formulation of a regular specification $R = f \circ g^\wedge$.
- ❇ Perform reification so as to preserve regularity, especially when input and output state spaces are distinct.
- ❇ If an operation subbornly refuses to be captured by a regular specification, strive to understand why. Is it fundamentally non-regular, or could an alternative approach (eg. a change in coordinates) bring out regularity? If so, is the change of perspective on the problem, cost-effective as an aid in development or is it best regarded as an aid in understainding? If no alternative view yields regularity, could alternative operations be designed which are regular, and if so, would they be more useful? Could the operation be broken down into smaller suboperations which are regular, and if so, is this insight helpful? If the former do not apply, would an analysis of the operation in terms of higher order permutability properties be helpful?

Fig. 3.

form because at a fundamental level, higher order permutability equations are needed to describe it. In that case a deeper study of the structure of the problem using the permutability properties might reveal aspects that can be exploited in implementations. However we have pursued these latter possibilities rather less in this paper and so this last suggestion must be on a more tentative level.

On the whole, we have amassed enough evidence to make plausible the claim that non-regular cases will be rare. Usually one will be able to find regularity, and then we recommend that it be used. It can help to structure the details of how a specification is developed, and this structure can be profitably exploited in the verification of the specifications developed. The author imagines that a structured and disciplined approach to the creation of a specification, such as is provided at least in part by regularity, will be particularly beneficial in computer-aided work, where it could lead to proof obligations which are rather more tractable than would otherwise be the case for reasons mentioned above. Further work would be needed to properly substantiate this though.

9 CONCLUSIONS

In the preceding sections of this paper, we have picked out the criterion of regularity of relations, a concept having deep universal properties, and shown that it has widespread applicability in specification design. As well as being manifest in deterministic specifications, we have shown it to be widely applicable to nondeterministic problems, and indicated that it has parallels with methods used in classical applied mathematics. As a result of this widespread applicability, we have recommended that the search for regularity be placed at the forefront of specification design, as being likely to lead to more understandable specifications, and ones that are easier to manipulate in subsequent stages of design. We have encapsulated our recommendations in a few slogans in the previous section.

We have stopped short of proposing a specific deveopment methodology based on the observations in ths paper. On the one hand this would take us outside the scope of the paper as stated in the introduction, on the other it would tend to emphasise whichever particular methodolgy we described in preference to others. This would be undesirable, since we intend the impact of this paper to be at the meta level: its ideas are capable of being brought to bear on many deveopment methodologies.

We have indicated that in the case of the few examples that defied easy description using regular relations, there is evidence to indicate that other relational metaphors might apply, based on higher order, or n-permutability. Material on this topic can be found in eg. Carboni et al. (1993). For $n = 3$ n-permutability corresponds to regularity, also called the Mal'cev property (Mal'cev (1954)). For $n = 4$ it is the Goursat property (Goursat (1889)). General values of n may well lead to a useful classification of relations arising in specification work.

Acknowledgements

It is a pleasure to thank Cliff Jones and Ian Hayes for discussions and comments on some of the material in this paper, and particularly for suggesting interesting examples of specifications with which to confront the regular methodology. Thanks are also due to Harold Simmons and Peter Johnstone for the Carboni et al. reference.

Appendix A: Some Deterministic Examples

For some more complex examples of deterministic specifications we refer to the collections Jones and Shaw (1990), and Hayes (1993). These abound with deterministic operations. From Jones and Shaw (1990), we might mention:

- ***delete_connection*** (from a database) in the NDB case study,
- ***count_tripple***, ***declare_verb***, ***delete_tripple***, ***grant_access***, ***initialise***, ***insert_tripple***, ***partition_clear***, ***test_verb***, ***undeclare_verb***, ***verb_inverse***, and others in the ISTAR case study,
- almost all the operations in the MUFFIN case study, (because they are already (partial) functions),
- ***dispose***, and some of the versions of ***new***, in the HEAP STORAGE case study,

❇ many of the operations in the GARBAGE COLLECTION case study,

and so on. From Hayes (1993) we might mention:

❇ the operations in the ***symbol table***, ***file update***, and ***sorting*** tutorials,

❇ the operations in the BLOCK STRUCTURED SYMBOL TABLE case study,

❇ the operations in the TELEPHONE NETWORK case study,

❇ ***readfile***, ***writefile***, ***createSS***, ***destroySS***, ***readCHAN***, ***writeCHAN***, ***seekCHAN***, ***closeCS***, ***readAS***, ***writeAS***, ***seekAS***, etc., in the UNIX FILING SYSTEM case study,

❇ the operations in the CAVIAR case study.

In fact the vast majority of operations mentioned in both collections of case studies are deterministic, as one might expect, and the vast majority of those that remain fit the canonical nondeterministic template, as listed below.

Appendix B: Some Canonical Nondeterministic Examples

Allocating a page in a heap-managed memory is the obvious canonical example, but there are many others. From Jones and Shaw (1990) we might mention:

❇ ***add_connection*** from the NDB case study,

❇ ***build_tripple*** from the ISTAR case study,

❇ ***spawn_proof***, and ***add_empty_proof*** from the MUFFIN case study,

❇ some versions of ***new*** from the HEAP STORAGE case study,

etc. From Hayes (1993) we might mention:

❇ ***openCS***, ***create*** from the UNIX FILING SYSTEM case study.

All of these can be recast as regular relations by using the techniques described in section 6. They also vary with regard to whether the assigned object u, is visible in the interface or not.

References

Banach R. (1994); Regular Relations and Bicartesian Squares, Theoretical Computer Science, **129**, to appear.

Carboni A., Kelly G.M., Pedicchio M.C. (1993); Some Remarks on Mal'cev and Goursat Categories, Sydney School of Mathematics and Statistics Report, **93**-19, Applied Categorical Structures, to appear.

Goursat É. (1889); Sur les Substitutions Orthogonales, Ann. Sci. Éc. Norm. Sup., **3** (6), 9-102.

Hayes I. J. (1993); Specification Case Studies (2nd ed.), Prentice-Hall.

Hayes I. J., Sanders J. W. (1993); Refinement With Input/Output Transformations, Working paper.

Hayes I. J., Jones C. B. (1989); Specifications Are Not (Necessarily) Executable, IEE Software Engineering J., **4**, 320-338.

Jaoua A., Mili A., Boudriga N., Durieux J. L. (1991); Regularity of Relations: A Measure of Uniformity, Theoretical Computer Science, **79**, 323-339.

Jones C. B. (1990); Systematic Software Development Using VDM (2nd ed.), Prentice-Hall.

Jones C. B., Shaw R. C. (1990); Case Studies in Systematic Software Development, Prentice-Hall.

Mal'cev A. I. (1954); On the General Theory of Algebraic Systems, Mat. Sbornik N. S., **35**, 3-20.

Mili A. (1990); An Introduction to Program Fault Tolerance, Prentice-Hall.

Mili A., Xiao-Yang W., Quing Y. (1986); Specification Methodology: An Integrated Relational Approach, Software Practice and Experience, **16**, 1003-1030.

Parnas D. L. (1992); Tabular Representation of Relations, Communications Research Laboratory, Report 260, Faculty of Engineering, McMaster University.

Ross K. A., Wright C. R. B. (1992); Discrete Mathematics, Prentice-Hall.

Schmidt G., Strohlein T. (1993); Relations and Graphs, Discrete Mathematics for Computer Scientists, Springer.

Spivey J. M. (1993); The Z Notation: A Reference Manual (2nd ed.), Prentice-Hall.

Suppes P. (1960); Axiomatic Set Theory, Dover (1972).

Tarski A. (1941); On the Calculus of Relations, J. Symbolic Logic, **6**, 73-89.