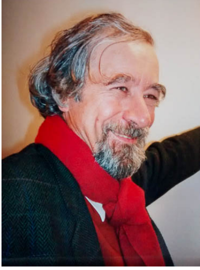
**Memories of Jean-Raymond Abrial
in Oxford and the Alps.**

Bernard Sufrin, Oxford, 2025



«Formalization is not an isolated activity, nor one which is finished once and for all. Implicitly at least it is a continuing aspect of every (good) designer's creative work, whether it is performed by a single individual, or what is often more fruitful, by a larger group, or even an entire community. In such situations, however, the (intermediate) results of this process should be given some form of concrete expression, whereupon they may be repeatedly examined, criticized, compared, and recast. It is in this context that the need for formal specification really arises. The notation employed must therefore be construed, above all, as a medium of communication.»[1]

Introduction

I collaborated very closely with Jean-Raymond while he was visiting the PRG[1] in Oxford from Autumn 1979 to mid 1981; and we remained good friends for many years afterwards. In this note I have interleaved a few personal and technical reminiscences of our friendship and collaboration.

My research brief from Tony Hoare when I arrived at the PRG in the autumn of 1978 was to explore ways of publishing the source texts of high-quality software. Academics in the then-tiny group had a lot to do that year: developing and teaching graduate courses, brokering collaborative projects with other parts of the University, *etc.* [2]. I spent some of my remaining time writing and re-writing a document compiler and a text editor[2] in a narrative style. In retrospect these attempts at "code as communication" would be called literate programs, but Knuth had yet to popularise the practice or the tools to support it.

I ended this year feeling uneasy about publishing these programs: they worked perfectly well but were they of sufficiently high quality to publish? The design principles behind the editor's user interface (modelessness and undoability) were lost in the details of its implementation. Something was certainly missing ... *but*

[1]The Programming Research Group of the Computing Laboratory. Until the 1990s this was the *de-facto* Computer Science Department of Oxford University.

[2]Why bother? Please remember that this all went on long before the days of `emacs`, and `word`; TeX was still over the western horizon, and computing facilities in the PRG were still somewhat primitive.

I didn't know what. Then Jean-Raymond arrived, and I began to understand what was wrong.

Finding reasonably priced accomodation for rent in Oxford was then – as it is now – notoriously difficult, but Jean-Raymond and his partner Hélène Villers had found a splendid house on Osney Island in Oxford a yard or so from the river Thames at the corner of South and East Streets. It had a studio-cum-drawing-office for Hélène to work in as she made the transition from academic sociologist to fine artist. I still treasure a few of her early works. The house had a garage that could fit two narrow saloon cars or two dozen bicycles, but J-R – a keen Alpinist – had brought his very large white "expedition van" and it had to be parked in the street outside. Fine until the floods hit Osney.

Specifications

Jean-Raymond's first course at Oxford was called "Program Specification". It was about using the language of sets, relations and functions to model software systems. It was different to any mathematics course I'd taken as an undergraduate: impressive because every new idea and notation he introduced was immediately illustrated by a couple of serious examples of its application to the description of a situation or system that meant something to the programmers present.

The flavour of his "Bourbakiesque" notation and approach, and some of his examples, can be seen at right and in figures 1-3 – all extracts from the paper that introduced his specification language, then only occasionally called Z [3].

```
SET =
  def
    op(∪)[X] = func S1,S2 → S3 for
                 S1,S2, S3 : subset(X)
               then
                 S3 = set x for x : X where
                        x ε S1 or x ε S2
                      end
               end;
    op(∩)[X] = func S1,S2 → S3 for
                 S1,S2,S3 : subset(X)
               then
                 S3 = set x for x : X where
                        x ε S1 and x ε S2
                      end
               end;
    union[X] = func SS → S for
                 SS : subset(subset(X));
                 S  : subset(X)
               then
                 S = set x for x : X where
                       exist S' for S' : SS where
                         x ε S'
                       end
                     end
               end;
```

Image of an extract from the SET chapter of "Specification Language"

There was much emphasis on the modular structuring of specifications: smaller theories, whether intended to be of general use or application-specific were organised as *chapters*, with inter-chapter dependencies made explicit.

What might be surprising from a contemporary point of view, and in view of Jean-Raymond's later work, was the absence of any *formal* rules of reasoning in the "Specification Language" paper. That is not to say that there were no proofs in the examples: just that the commonsense (!) laws of set theory and its derived theories were taken for granted.

```
REL =
 use SET def
   inv[X,X']     = func r → r' for r : X ↔ X'; r': X' ↔ X then
                     r' = rel x' ↔ x for x' : X'; x : X where
                            r(x ↔ x')
                          end
                 end

   op(∘)[X,Y,Z] = func r2,r1 → r3 for
                          r1 : X ↔ Y; r2 : Y ↔ Z; r3 : X ↔ Z
                 then
                   r3 = rel x ↔ z for x : X; z : Z where
                          exist y for y : Y where
                             r1(x ↔ y); r2(y ↔ z)
                          end
                        end
                 end
   ...

   functional[X,Y] = set r for r : X rel Y where
                         r'(Y) = X; (r ∘ r') ⊂ ident[Y]
                     given
                         r' = inv(r)
                     end;

   function[X,Y]   = func r → f for r : X ↔ Y; f : X → Y where
                        r ∈ functional[X,Y]
                     then
                        f = func x → y for x : X; y: Y then
                               y = any(R(x))
                            end
                     end
```

Figure 1: An extract from the REL chapter

```
SEQ =
 use SET, NAT, REL def
   segment              = func n → S for n : NAT; S: subset(NAT) then
                              S = set i for i : NAT where i<n end
                            end;
   ...
   seq[X]               = set s for s : NAT ⇸ X where
                              dom(s) ∈ segment(NAT)
                            end;

   length[X]            = ...

   op(*)[X]             = func s1,s2 → s3 for s1,s2,s3 : seq[X] then
                              s3 = s1 ∪ (s2 ∘ f)
                             given
                              f = iter(inv(succ))(length(s1))
                             end;
   ...
   associated_rel[X]    = func s → r for
                              s : seq[X];
                              r : X ↔ X
                             then
                              r = s ∘ succ ∘ inv(S)
                             end
```

Figure 2: An extract from the SEQ chapter

The definitions here are typical of Jean-Raymond's adoption of the extensional approach to relations and functions in his constructions. For example, the *associated relation* of a sequence is what we would now call its adjacency relation: it relates elements that are directly adjacent in the sequence. Likewise, the description of the result of sequence catenation ($*$) as the union (qua relation) of $s1$ and "$s2$ right shifted by the length of $s1$". The proof that the resulting relation is a function whose domain is a `segment` is straightforward.

For his first "realistic"[3] example, Jean-Raymond shows the specification and development (over four A4 pages) of a straightforward program that replaces multiple adjacent blanks in its input by a single blank. What may be of interest here is the use of classes and inheritance in the specification, and the incremental description of the specification itself.

```
EDITING_PROBLEM =
  use RELATIONS, MINI_FIXED_POINT_THEORY def
     state[C]        = class
                         b : C;
                         in, out : seq[C]
                       end;

     spec1[C]        = subclass state[C] where
                         out ∈ no_two_consecutive_blanks
                       given
                         no_two_consecutive_blanks =
                            set s for s : seq[C] where
                              not(associated_rel(s)(b ↔ b))
                            end
                       end;

     spec2[C]        = subclass spec1[C] where
                         equivalent_string(in ↔ out)
                       given
                         equivalent_string = closure(r);
                         r = rel s1↔ s2 for s1, s2: seq[C]
                             where
                               exist x,y,b1,b2 for
                                 x,y : seq[C]
                                 b1,b2 : seq[{b}] - null
                               where
                                 s1 = x*b1*y
                                 s2 = x*b2*y
                               end
                             end
                      end
```

Figure 3: An extract from the EDITING_PROBLEM chapter

Here spec1 is the class of states whose output has no two consecutive blanks; and spec2 is the subclass of that class in which the output and input are equivalent modulo (nonempty) sequences of space.

The details of the subsequent development from specification to program are of interest but constrained space precludes longer discussion here, save to observe that there is no "end to end" specification that captures the *requirement* that the final out is the "shortest equivalent_string to the (initial) in." To remedy this readably would almost certainly give rise to a refactoring of the material of this chapter.

[3]Abrial's scare-quotes!

Towards "Oxford Z"

Early the following year Jean-Raymond and I, with our new postdoc (Tim Clement), and Ib Sörensen (a D.Phil student who had responded very enthusiastically to the Program Specification course), began work on the CAVIAR[4] project. It had been generously funded by **S**tandard **T**elecommunications **L**aboratories.[5] Its goal was to evaluate the effectiveness of formal specification techniques in communicating ideas about "realistic systems" by specifying a system to be used by the company to manage room scheduling and resource provision for the (very many) visitors to its buildings.

The four of us spent the next few months refining the existing specification notation, and thinking about CAVIAR with it. We'd meet every afternoon in the smoke-filled office occupied by Jean-Raymond, Tim, and Ib. It was a heady time: when we weren't changing the specification [of CAVIAR] we'd change the language. Jean-Raymond had the big language ideas, and wrote them up frighteningly fast, but he wasn't dogmatic, and he listened to everybody's ideas. Ib Sörensen took responsibility for writing up the CAVIAR specification itself [5], and Jean-Raymond and I experimented with using the notation for other applications, and started to prepare a graduate course on specification we would give jointly at the start of the next academic year.[6]

What emerged from our deliberations was the "second generation Z" [1, 4]: still rooted in a generically-typed set theory and with "basic library" still presented in the "layered" style of Bourbaki.

By way of illustration I have shown the whole of the `SET` chapter in figures 4 and 5. In figures 7 and 6 I have also shown a handwritten note by Jean-Raymond experimenting with one incremental style of presentation and the opening chapter of my own attempt at an editor specification.

What has changed is that chapters themselves are generic, functions and relations are now given explicit (type-) signatures separated from their definitions, and intended or proven properties of the declared objects are given.[7] A new (obviously unoriginal) idea was that one could present an application-oriented theory as a signature together with intended properties whilst deferring the eventually-necessary constructive definitions. A few pedants didn't like this, and would quote Bertrand Russel at us: *"The method of 'postulating' what we want has*

[4] Computer Assisted Visitor Information And Resources

[5] Enthusiastically advocated within STL by Bernard Cohen and Tim Denvir.

[6] The MSc cohort to which it was delivered is pictured in [2](p30)

[7] For example, the chapter `REL` on relations has fourteen collections of properties of the relational operators defined in it.

many advantages; they are the same as the advantages of theft over honest toil." I'll leave you to imagine our stock riposte.

We sent our working papers to Dana Scott, an old friend of the group who was then Professor of Mathematical Logic in the sub-Faculty of Philosophy. His kindly note in response deflated us somewhat. He liked our goals for the language, but not its long windedness: "... and when it takes you nearly half a page to write down an existential over a simple structure... you should perhaps think of adopting more conventional notations in some places."

We resolved to deal with the prolixity; but another of his remarks "why don't you make all your specifications implementable by restricting your language to ... [continuous functions over domains]" was echoed elsewhere. We would respond by saying that we *wanted* our users to be able to specify the unimplementable: in such cases the process of refinement would uncover the unimplementability and cause the specifiers (and their clients) to think again.

"Oxford Z" begins to emerge

Our rethinking of the notation first necessitated that we detach ourselves from the idea that we could somehow make mathematical ideas more palatable to working programmers by packaging them in a notation that resembled a programming language.[8]

It was Héleǹe who provided the inspiration behind the "box and lines" style of presentation that became the (superficial) trademark of a Z specification. More importantly, though, we reverted to a recogniseably more orthodox presentation of mathematics: and spent significant effort on systematising it. Most notably we unified the notations for introducing and constraining variables – whether in the signatures of theories, the bodies of structures, or the bound variables of quantified constructs. The ability to name a signature/predicate structure that this liberated gave rise to the notion of a schema.

Comparing figure 8 with figure 7 may give an inkling of the simplifications we had made in this phase, and the firming up of our view that a "state" was really just a collection of related observations, some of which might never appear in the "reified"(Jones) data of an implementation. The simplified notation was promulgated quite widely: especially in the PRG and at INRIA [7]. Discussion of its later evolution and the incorporation of a *calculus* of schemas is beyond the scope of this note.

[8] I remember a couple of painful sessions of *autocritique* over dinner in Jean-Raymond and Héleǹe's kitchen. The three of us had dinner together very frequently during this period for reasons that I allude to later.

```
SET[X] ≙
chapter with

   This Basic Chapter defines the usual set Operators as well as their classical
   properties.

   op(⊂), op(⊄)  :  subset(X) ↔ subset(X) ;

   op(∪), op(∩), op(−)  :  subset(X) × subset(X) ↔ subset(X) ;

   union, inter  :  subset(subset(X)) → subset(X)

   partition, subset1  :  subset(subset(X)) ;

def

   op(⊂) ≙ rel S1 ↔ S2 where S1 ∈ subset(S2) end;

   op(⊄) ≙ rel S1 ↔ S2 where not(S1 ⊂ S2) end;

   op(∪) ≙ fn S1,S2 → S3 then S3 ≙ set x where x ∈ S1 or x ∈ S2 end end;

   op(∩) ≙ fn S1,S2 → S3 then S3 ≙ set x where x ∈ S1 and x ∈ S2 end end;

   op(−) ≙ fn S1,S2 → S3 then S3 ≙ set x where x ∈ S1 and x ∉ S2 end end;

   union ≙
       fn SS → S then
           S ≙ set x where exist S1 for S1 : SS where x ∈ S1 end end
       end;

   inter ≙
       fn SS → S then
           S ≙ set x where all S1 for S1 : SS imply x ∈ S1 end end
       end;

   partition ≙
       set SS where
           union(SS) = X;
           all S1,S2 for S1,S2 : SS where S1 ≠ S2 imply null(S1 ∩ S2) end
       end;

   subset1 ≙ subset(X) − {null[X]};
```

Figure 4: First part of the SET chapter of the Basic Library

```
prop ≙
    th A,B,C for A,B,C : subset(X) imply
        A ⊂ B and B ⊂ C   ⟹   A ⊂ C;
        A ⊂ A;
        A ⊂ B and B ⊂ A   ⟺   A = B;

        A ∪ A = A;
        A ∪ B = B ∪ A;
        A ∪ (B ∪ C) = (A ∪ B) ∪ C;
        A ∪ null[X] = A;
        A ∪ X = X;

        A ∩ A = A;
        A ∩ B = B ∩ A;
        A ∩ (B ∩ C) = (A ∩ B) ∩ C;
        A ∩ null[X] = null[X];
        A ∩ X = A;

        A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C);
        A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C);

        X - (X - A) = A;
        (A ⊂ B) ⟺ (X - B) ⊂ (X - A);
        A ∪ B = X - ((X - A) ∩ (X - B));
        A ∩ B = X - ((X - A) ∪ (X - A));
        A - B = A ∩ (X - B);

        union({A,B}) = A ∪ B;
        inter({A,B}) = A ∩ B
    end

end SET
```

Figure 5: Second part of the SET chapter of the Basic Library

In this example, I try to develop incrementally the "description" of a small system, by using the "class" and "class_Fn" notation.

Let us define a "basic file handler" as a class containing two components:

- a file (described as a function from keys (K) to records (R))
- a set of private buffers (described as a function from processes (P) to records)

```
basic_file_handler [K,R,P] ≙
  class with
  | file : K → R;
  | buffer : P → R
  end
```

We now provide two class_Fns for "reading" or "writing". They both have the same shape given by

```
basic_cmd [K,R,P] ≙
  class_Fn identity [basic_file_handler[K,R,P]] with
  | key : K;
  | process : P
  end;
```

We may now define

```
basic_read [K,R,P] ≙
  class_Fn basic_cmd [K,R,P] def
  | buffer ≙ buffer ⊕ {process → file(key)}
  end;
```

```
basic_write [K,R,P] ≙
  class_Fn basic_cmd [K,R,P] def
  | file ≙ file ⊕ {key → buffer(process)}
  end;
```

Figure 6: Extract from an experiment in defining a file handler

First we describe the basic class which is used to represent the state of the editor. The class is generic with respect to X -- which should be interpreted as the intended character-set of the editor.

```
CUT[X] ≙
chapter
        SEQ[X]
type
        cut ≙
            class with
                file, left, right: seq[X]
            def
                file ≙ left * right
            end
```

The state of the file being edited is represented by two sequences -- that comprising the text which preceeds the cursor, and that comprising the text which follows the cursor.

```
with
        delete, move    : cut ⇸ cut;
        insert          : X → (cut → cut)
def
        delete ≙
            cfn where
                right ≠ null[seq[X]]
            then
                right ≙ tail(right)
            end

        move ≙
            cfn where
                right ≠ null[seq[X]]
            then
                left ≙ left * <first(right)>;
                right ≙ tail(right)
            end

        insert ≙
            fn ch → f then
                f ≙
                    cfn cut → cut then
                        left ≙ left * <ch>
                    end
            end
end CUT
```

At the moment we do not have enough material to define an editor since we cannot "undo" an insertion or move backwards

Figure 7: Part of the "Display Editor" in "Z"

Let X be the intended character set of the editor.

The CUT schema captures the three observations that can be made of the "file being edited": the file itself, and the texts that preceed and follow the "current editing position". Notice that any two of these observations are enough to capture the third: we exploit this to give abbreviated definitions of the simplest operations of the editor.

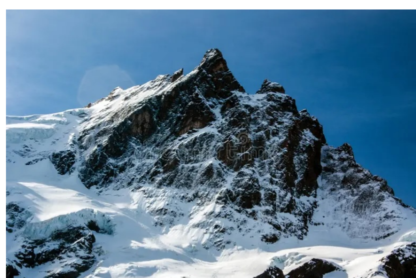
$$\begin{array}{|l}
\textit{CUT} \\ \hline
\textit{file}, \textit{left}, \textit{right} : \mathrm{seq}\, X \\ \hline
\textit{file} = \textit{left} \frown \textit{right} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\textit{delete}, \textit{move} : CUT \nrightarrow CUT \\
\textit{insert} : X \longrightarrow CUT \longrightarrow CUT \\ \hline
\textit{delete} = \\
\quad \lambda\, CUT \mid \textit{right} \neq \langle\rangle \bullet \\
\quad\quad \mu\, CUT' \bullet \\
\quad\quad\quad \textit{right}' = \textit{tail}(\textit{right}) \\
\quad\quad\quad \textit{left}' = \textit{left} \\
\\
\textit{move} = \\
\quad \lambda\, CUT \mid \textit{right} \neq \langle\rangle \bullet \\
\quad\quad \mu\, CUT' \bullet \\
\quad\quad\quad \textit{left}' = \textit{left} \frown \langle \textit{first}(\textit{right}) \rangle \\
\quad\quad\quad \textit{right}' = \textit{tail}(\textit{right}) \\
\\
\textit{insert}(x) = \\
\quad \lambda\, CUT \bullet \\
\quad\quad \mu\, CUT' \bullet \\
\quad\quad\quad \textit{left}' = \textit{left} \frown \langle \textit{first}(\textit{right}) \rangle \\
\quad\quad\quad \textit{right}' = \textit{tail}(\textit{right})
\end{array}$$

Figure 8: Part of the "CUT" chapter of the "Display Editor" in early "Oxford Z"

Alpine Adventures

During the summer of 1980, while I was recovering from the end of a personal relationship, I stayed with Jean-Raymond and Hélène for a few weeks in a borrowed chalet up by the old Olympic downhill course. We did a lot of scrambling together – usually after long bumpy drives along the D1091. Happily there was room for three of us in the front of the white van, for the mountain scenery was never less than spectacular; but when a fourth joined us somebody had to rattle about with the equipment in the back.



Once we ascended the foreslopes of La Meije from the direction of Deux Alpes. After about four hours scrambling, and long before we had planned to descend, I was bitten horribly behind the knee by a huge horsefly. My whole leg swelled up enormously in about 15 minutes! J-R and our other companion had to help me hobble the long route back down with a nonfunctioning leg. We had to endure a difficult few hours descent before there would be a chance of getting it seen to; and my boot had to be cut off before we reached the van.

Eventually, after the long drive back to Grenoble, we reached a hospital. But it was 15th August, "the Assumption" – a day when the contradictions between the formally secular nature of the French state, and the Catholic nature of a significant proportion of the French have been historically resolved by making it a day when everything remotely public must close. To my dismay the toxicologists were on holiday with "Adieu toxico', via Mexico!" on their whiteboard, but somebody gave me some antihistamine to stem the awful swelling, and by the next day it had been replaced by ... a pervasive itch.

Another time, on Petit Taillefer, we had to rescue an inexperienced youth who had fallen on a rocky pitch that was much too hard for him. His companions, rich, elderly[9] gentlemen: were scarcely climbers at all. The rescue required us to carry the youth down the mountain for about 6km.

[9] it seemed to me then!

Jean-Raymond did most of the carrying on his back, whilst I grunted obligingly in the rear, carrying the youth's equipment.

Au revoir

I wrote earlier of Jean-Raymond's frighteningly fast work rate. And for the first few months of 1981 he worked even more quickly – generating large numbers of drafts of new foundations for the language and its basic library, as well as building the prototype of an extensible proof checker, and writing essays inspired by Cliff Jones's book[6]. This work would (eventually) become the basis for **B** and the **B tool**. The rest of the "Z group" found it impossible to keep up with him, and wanted the notation to stay stable for a while. We had materials to prepare for courses and further collaborations with industrial partners to pursue,[10] and a large (5 year) Software Engineering project to prepare for.

Then one Friday lunchtime, Hélène Villers appeared without Jean-Raymond at Brown's restaurant in Oxford to join Cliff and Jill Jones, Linda Forrest and me, and Jill Hoare for lunch. She brought the upsetting news that Jean-Raymond had unilaterally decided that they would both leave Oxford and return to Paris at the end of the following month.[11] She didn't know why, and despite us all remaining good friends with them both for years afterwards none of us ever found out.

> "Those who have the privilege of friendship with Jean-Raymond Abrial have long been aware of the great work in which he has been engaged. It is no less than a complete understanding of the nature of software engineering: ..."
> (from C.A.R.Hoare's prefatory tribute to Abrial in "The B-Book")

[10] Ib Sörensen had started working with the CICS team at IBM on a project to develop a rational reconstruction of the ideas behind this large, long-lived, and difficult-to-maintain transaction processing system.

[11] At least they had somewhere to live. They had retained Hélène's student-era bathroomless "Chambre-de-bonne" apartment 6 floors' walk upstairs in a town-house near Place Denfert-Rochereau.

References

[1] Jean-Raymond Abrial
The Specification Language Z: Syntax and “Semantics”
Programming Research Group Working Paper, April 1980

[2] Bernard Sufrin
Early Days at the PRG ... anecdotes from the Programming Research Group for Tony Hoare’s 90th birthday.
FACS FACTS, 24-2, July 2024

[3] Abrial Jean-Raymond; Schuman, Stephen A; Meyer, Bertrand
A Specification Language
in On the Construction of Programs, in MacNaghten, A. M.; McKeag, R. M. (eds.) Cambridge University Press (1980), ISBN 0-521-23090-X.

[4] Jean-Raymond Abrial
The Specification Language Z: Basic Library
Programming Research Group Working Paper, April 1980

[5] Jean-Raymond Abrial, Bernard Sufrin, Tim Clement, Ib Sörensen
Specification of CAVIAR
a Computer Aided Visitor Information and Reception System
Programming Research Group Working Paper (undated)

[6] Cliff B. Jones
Software Development: A Rigorous Approach
Prentice-Hall International (Jan. 1980)
ISBN 978-0138218843

[7] Bernard Sufrin
Formal System Specifications (Notation and Examples)
in Tools and Notions for Program Construction an advanced course
D. Néel (ed). Cambridge University Press, Jan 1982
ISBN 0-52124801-9

Note

All technical material in this note is either exactly transcribed from papers that sat in my personal archive for many years before being scanned, or a screen shot of part of such a scan. The scans are available on my Oxford website.