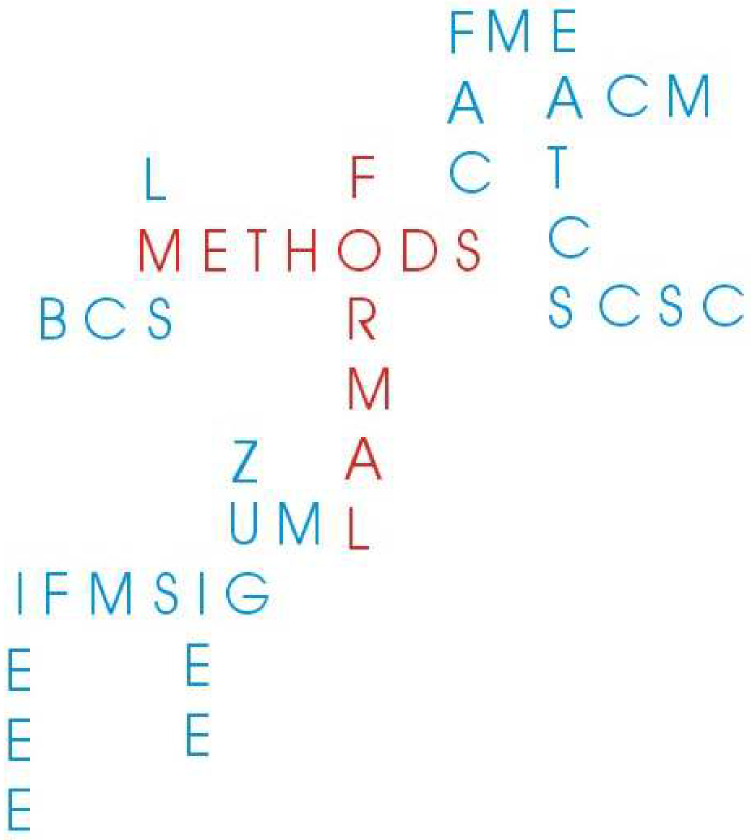
Issue 2022-2
July 2022

FACS FACTS



Formal Aspects of Computing Science Specialist Group

The Newsletter of the Formal Aspects of Computing Science (FACS) Specialist Group

ISSN 0950-1231

About FACS FACTS

FACS FACTS (ISSN: 0950-1231) is the newsletter of the BCS Specialist Group on Formal Aspects of Computing Science (FACS). FACS FACTS is distributed in electronic form to all FACS members.

Submissions to FACS FACTS are always welcome. Please visit the newsletter area of the BCS FACS website for further details at:

https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/

Back issues of FACS FACTS are available for download from:

https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/back-issues-of-facs-facts/

The FACS FACTS Team

Newsletter Editors

Tim Denvir	timdenvir@bcs.org
Brian Monahan	brianqmonahan@googlemail.com

Editorial Team:

Jonathan Bowen, John Cooke, Tim Denvir, Brian Monahan, Margaret West.

Contributors to this issue:

Jonathan Bowen, Tim Denvir, Brian Monahan, John Tucker

BCS-FACS websites

BCS:	http://www.bcs-facs.org
LinkedIn:	https://www.linkedin.com/groups/2427579/
Facebook:	http://www.facebook.com/pages/BCS-FACS/120243984688255
Wikipedia:	http://en.wikipedia.org/wiki/BCS-FACS

If you have any questions about BCS-FACS, please send these to Jonathan Bowen at jonathan.bowen@lsbu.ac.uk.

Editorial

Dear readers,

Welcome to issue 2022-2 of the *FACS FACTS* newsletter. This is our mid-year issue for 2022.

In this issue we have two Features. An article by John Tucker giving a historical view of operational semantics, particularly as applied to the semantics of PL/1. This formal semantics was developed at the IBM Vienna Laboratories in the second half of the 1960s. John gives an honest, considered and eloquent account of the formalisation work and of the concepts driving the design of PL/1, including their subsequent criticism.

Brian Monahan's "An Awkward Problem" comprehensively illustrates a seemingly straightforward program design task: it shows how mathematics is pervasive and necessary in designing algorithms, and how algorithms need to be understood as mathematical abstractions of program behaviour. You could call it an object lesson for a formal approach.

It is ten years since Ib Holm Sørensen passed away at the early age of 62, in the centenary year of the birth of the computer science pioneer Alan Turing. Jonathan Bowen writes an appreciation of his life and achievements.

Jonathan also commemorates thirty years since the formation of the Z User Group. The Z User meetings produced numerous proceedings in the Springer Workshops in Computing and LNCS series, the covers of which are shown.

Finally, a review by Tim Denvir of John Barnes' (of *Ada* and *RTL/2* fame) book, *Nice Numbers*. Everyone who delves into formal aspects of computing comes to grips with quite a lot of mathematics, much of which is outside the traditional curricula. Despite being about numbers, which we all think we know about, this book contains a lot which will be new and fascinating even to those well versed in mathematics.

As we said in the editorial of issue 2022-1 (see bcs.org/media/8289/facs-jan22.pdf), we very much appreciate and look forward to contributions, especially comments, from you, our readers.

We hope you enjoy *FACS FACTS* issue 2022-2.

Tim Denvir
Brian Monahan

Table of Contents

History of Computing Collection at Swansea University

The History of Computing Collection specialises in computing before computers, formal methods, and local histories of computing. An introduction to the Collection appeared in the February 2021 issue of *FACS FACTS* (2021-1, pp.10-17). The Collection is located on the Singleton Campus of Swansea University; it can be visited by appointment. A small number of items from the Collection are on display in the Computational Foundry, Bay Campus, which is the home of the Computer Science Department. All inquiries welcome.

From the History of Computing Collection, Swansea University:

PL/1 in New York, Winchester and Vienna

John V. Tucker
Swansea University

Operational semantics is truly basic in the theory of programming and programming languages. The idea is simple enough: the semantics of a program is characterised by modelling its behaviour, i.e., what it does. The things to be modelled are (i) states of a machine and (ii) transitions from one state to another. How hard can that be? Surely, anything can be modelled using idealisations and abstractions to postpone difficulties and eliminate inessentials.

Dream on. Machines are already idealisations of a mass of complex functional components, and programs are full of constructs that singly or in combination can generate obscure and unforeseen actions. Thus, the answer is that it is awfully hard and, indeed, since the 1960s all sorts of semantical approaches to the 'meaning' of programs have been developed for all sorts of computational situations and needs.

But why bother? This audience has a number of answers to that – e.g., modelling helps

- understand design choices and decisions and predict their consequences;
- improve languages and tools that improve programs and programming.

There are those of us who simply enjoy semantic modelling as their means of exploring the behaviour of data and computations. But, crucially for our history, modelling helps

- form a basis for precise specifications that are sufficiently abstract to make languages and programs portable between machines.

There is a distinction to be made between modelling and formalising – (i) & (ii) versus (iii) – already evident in Bertrand Russell's views on the formalisation of mathematics. But that topic is for another occasion.

In the folk history of programming languages, if PL/1 appears as a milestone then it is in the formal development of language specifications. It marks a step forward in programming semantics after the achievement of Algol for programming syntax. Indeed, the significance of PL/1 is 'reduced' to that of operational semantics, which is associated with the IBM Vienna Laboratory.

Of course, the language deserves much more historical attention. Anyway, the formal methods community must keep its memory alive and cherish it, as we do Algol. The centre piece for today's choice from the Collection is the set of IBM Vienna reports on PL/1 in **Figure 1**.

The Vienna Reports on PL/1

The formal definition of PL/1 by the Vienna Lab comes in three 'published' versions, in December 1966, June 1968 and April 1969. The reports in Figure 1 are those of the second version, all released on 28 June 1968. These reports, and most of the other Vienna Lab Reports in our Collection, are a small part of a gift to the Collection by Dines Bjørner, a scientist who needs no introduction to this audience. Dines worked in IBM 1962-75 and at the Vienna Lab 1973-75.

The reports in Figure 1 are these:

P. Lucas, K. Alber, K. Bandat, H. Bekic, P. Oliva, K. Walk and G. Zeisel. *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*. Technical Report 25.083. IBM Laboratory Vienna, 1968.

K. Alber and P. Oliva. *Translation of PL/I into Abstract Syntax*. Technical Report 25.086. IBM Laboratory Vienna, 1968.

K. Alber, P. Oliva and G. Urscler. *Concrete Syntax of PL/I*. Technical Report 25.084. IBM Laboratory Vienna, 1968.

M. Fleck and E. Neuhold. *Formal Definition of the PL/I Compile Time Facilities*. Technical Report TR 25.080. IBM Laboratory Vienna, 1968.

K. Walk, K. Alber, K. Bandat, H. Bekic, G. Chroust, V. Kudielka, P. Oliva, and G. Zeisel. *Abstract Syntax and Interpretation of PL/I*. Technical Report TR 25.082. IBM Laboratory Vienna, 1968.

Together they define PL/1, correcting and updating the Lab's first attempt two years earlier. The following year a last updated version was published, with the same titles and largely the same authors.

Let's look at the birth of PL/1 and reflect on the achievement.

IBM in the late 1950s

PL/1 is a part of the legacy of IBM's System 360 product line, announced on 7 April 1966. System 360 is an achievement in the history of practical computing, one with great technical and financial risks and rewards for IBM. The decade before was rather dramatic for IBM. It began with the transfer of power from Thomas J Watson Sr to his son Thomas J Watson Jr, whose vision it was to make computers core to the business, and who transformed the operating structures of the company. Technically, it saw two transformations: (i) the transition from tabulating equipment to computers and (ii) the transition from a disparate incompatible set of computers to the compatible System 360 series. In both cases, the transitions involved abandoning products that were hugely profitable and primary sources of IBMs huge revenues. Business history was being made, as documented in Cortada (2019).

So, what was the problem that needed Watson Jr to bet the company? It was his customers' need for compatible machines and portable software. The transition away from electro-mechanical to electronic large-scale data processing was well established in the late 1950s, but this also meant that it was growing a market for more powerful equipment. Up-grading to more powerful computer systems was natural, but was unnaturally hard. IBM's entry level 1401 series was selling well, but buying a new more powerful machine meant re-programming software and re-training staff. When the need for more processing arrived, companies faced unwelcome costs, both if they stayed with IBM or migrated to another firm. Buying more 1401s meant other problems (such as maintaining consistent data bases on machines that did not communicate). So the vision was a completely new product line with compatibility and communication between machines and, for good measure, completely new physical and software

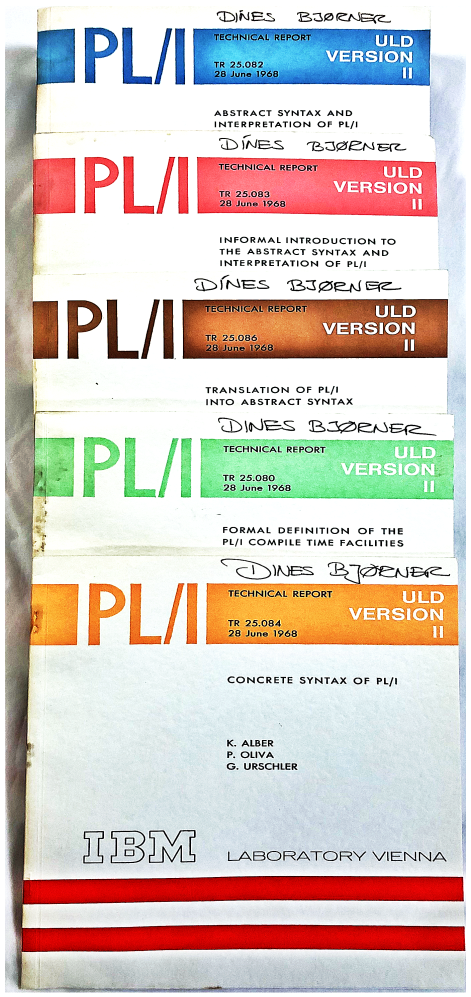

DINES BJØRNER
PL/I TECHNICAL REPORT ULD VERSION II
TR 25.082 28 June 1968
ABSTRACT SYNTAX AND INTERPRETATION OF PL/I
DINES BJØRNER
PL/I TECHNICAL REPORT ULD VERSION II
TR 25.083 28 June 1968
INFORMAL INTRODUCTION TO THE ABSTRACT SYNTAX AND INTERPRETATION OF PL/I
DINES BJØRNER
PL/I TECHNICAL REPORT ULD VERSION II
TR 25.086 28 June 1968
TRANSLATION OF PL/I INTO ABSTRACT SYNTAX
DINES BJØRNER
PL/I TECHNICAL REPORT ULD VERSION II
TR 25.080 28 June 1968
FORMAL DEFINITION OF THE PL/I COMPILE TIME FACILITIES
DINES BJØRNER
PL/I TECHNICAL REPORT ULD VERSION II
TR 25.084 28 June 1968
CONCRETE SYNTAX OF PL/I
K. ALBER
P. OLIVA
G. URSCHLER
IBM LABORATORY VIENNA

Figure 1: ULD - II

technologies. The days of the extremely profitable current product range were numbered.[1]

The press release on the 7 April 1964, summarised admirably the new features and advantages of System 360. There was a wink in the direction of its programming languages in the form of System 360's *applicable versatility*:

> The traditional distinction between computers for commercial and scientific use is eliminated in System/360. Users will be able to process both business and scientific problems, or a combination of the two, with equal effectiveness. (IBM Announcement 1964).

For some of those among the 100,000 businessmen in 165 American cites at which System/360 was announced this must have meant something to do with the future of Fortran and Cobol.

The conception of PL/1

PL/1 saw the light of day in the computer science community in January 1965 in an article in the *Communications of the ACM*, Radin and Rogoway (1965). Then, it was called *NPL* for *New Programming Language*, which resonated with IBM's new product line. NPL began in October 1963 when IBM and, specifically, SHARE created an *Advanced Language Development Committee* with certain aims to define a new language. SHARE was a user community for IBM products, run by volunteers. It was founded in 1955 by users of IBM 701s in Los Angeles and had grown, become organised, influential, and generally interested in IBM products, hard and soft.[2] Its origins and culture was close to Fortran. SHARE continues today.[3]

By the time of the publication, the *Advanced Language Development Committee* had seven members from major corporations and five from IBM; all were experienced technical people. The aims set by SHARE were to make a language that:

- satisfies the needs of a wide range of programmers;
- takes a simple approach to reduce programming errors;
- suits the development of the latest applications. Radin and Rogoway (1965).

[1] The 1401s lived from 1959 to 1970.

[2] The 701 is IBM's first commercial scientific computer, launched in 1952.

[3] See: https://www.share.org

The languages they had in mind were Fortran, COBOL and the *tour de force* that was Algol, in which IBMers had been heavily involved. It was an IBMer and Fortran creator, John Backus, who introduced the BNF method for formally defining the programming language syntax of Algol.

Not unlike the division in machines mentioned earlier, software divided into three categories: scientific and engineering, business and financial, and real-time processing and systems. Developers had settled for different programming languages, especially the Fortran and COBOL of the day. NPL was expected to be able to replace them.

Early on, as soon as computers arrived in organisations, both kinds of users emerged. For example, in Glamorgan, when a Ferranti Pegasus was purchased for the operations research group to design processes for the new steelworks at Port Talbot, it was not long before administration found things for it to do. The same happened when Swansea University bought its first machine, an IBM 1620 for scientific and engineering simulation (Tucker 2020). Convergence and portability made perfect technical and economic sense.

Along with convergence issues for machines came the need for convergence for programming constructs – constructs that were found to be desirable in one current language but were not present in another might be included. From the beginning, the solution of NPL was to acquire constructs.

Development of PL/1

From October 1963, the development of PL/1 has many milestones and the complete documentation for PL/1 is large. It is also dispersed. The technical development of the language was passed to IBM's Hursley Lab, near Winchester. This meant defining the language precisely enough for the construction of a compiler – a highly demanding and fundamental next step. Vienna's early interest in debating the language was rewarded by securing the task of making its formal definition – a great challenge and a plum assignment for the Lab, though dependent on Hursley, of course. Clearly, the aim of PL/1 and System 360 was to advance computing practice in a historically significant way.

The Vienna Lab at the time of the announcement of System 360 was considered 'special'. As a Development Lab and part of IBM Austria it was not part of IBM Research, and seen as in need to attention: it was on offer to Ambros Speiser as part an incentive package for him to remain head of IBM Zurich – a jewel in the crown of IBM Research – rather than leave (which he did: Speiser 1998).

The major documents for the three phases are:

1 the *System Reference Document* (SRD) of November 1964, owned by New York;

2 the semiformal definition of 1966, owned by Winchester; and

3 the three versions of the formal definition of the Universal Language Description (ULD) of 1966-69, owned by Vienna.

For convenience, these are called *ULD I*, *ULD II* and (the three versions of) *ULDIII*. These notations can be a little confusing (because of the appearance of the Vienna Reports cf. Figure 1) and were cooked up by Vienna and Hursley.

The job was not over in 1969. The work on PL/1 give rise to the general *Vienna Definitional Method*, and other languages, such as Fortran and Algol, were defined formally, as in **Figure 2**. A few years later there is the later specification in **Figure 3**, a key report for the general Vienna language definition method:

> Hans Bekic, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. *A Formal Definition of a PL/I Subset*. Technical Report 25.139, IBM Laboratory, Vienna, 1974.

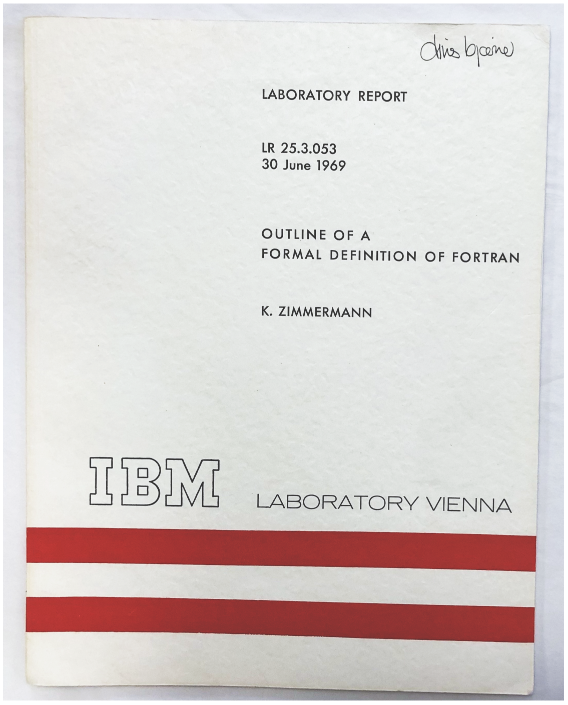
Subsequently, PL/1 was to receive a standardisation from the American National Standards Institute (ANS Programming Language PL/I. X3.53-1976).

Here, I will keep away from the build-up of many new features and constructs for the language, the emergence of the specifications at Hursley and Vienna, and the intricacies of the versions. There are contemporary introductions such as Lucas and Walk (1969) and Beech (1970); and there are later reflections, such as Radin (1978) and Lucas (1981); Radin’s was expanded upon in the PL/1 Session in Wexelblat (1981). And there is the scholarly work of Cliff Jones on the contributions of Hans Bekic (Jones 1984). My Swansea colleague Troy Astarte has tackled some of these technical matters for PL/1, and far more of the history, in Chapter 5 of his Newcastle PhD (Astarte 2019), which I recommend.

From the beginning, starting with the early views of SHARE, the language was seen as complicated … too complicated for some. This view grew as the language was discussed outside IBM in professional meetings, attracting critics such as Edsger Dijkstra and Tony Hoare whose view of programming and programming languages emphasised conceptual understanding and reasoning – and became an orthodoxy of the academic community for a generation. Surely, for that vision, PL/1 was to be seen as an example of how *not* to make a programming language:

> One of my implicit morals will be that such programming languages, each in their own way, are vehicles inadequate to guide our thoughts. If FORTRAN has been called an infantile disorder, PL/1 must be classified as a fatal disease.

(Dijkstra 1971, see also Dijkstra 1970).


[illegible]

LABORATORY REPORT

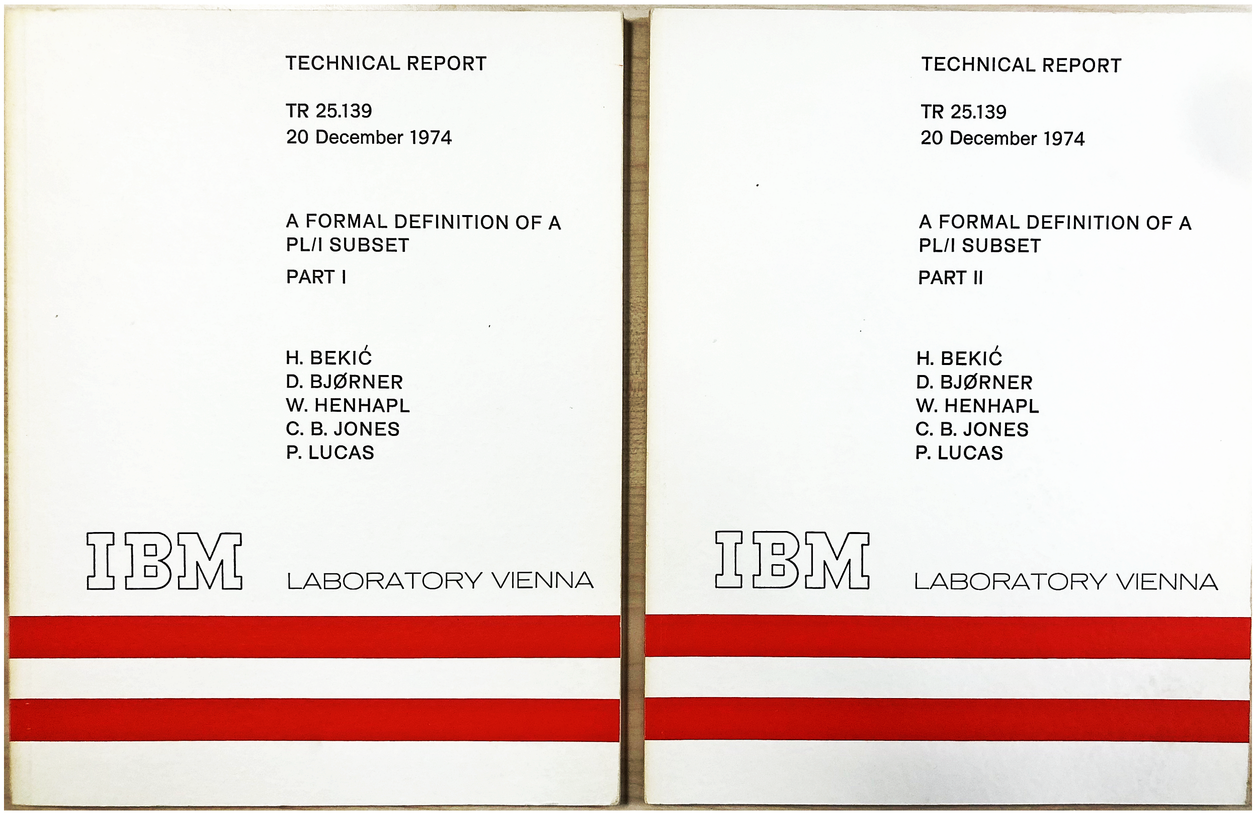
LR 25.3.053
30 June 1969

OUTLINE OF A
FORMAL DEFINITION OF FORTRAN

K. ZIMMERMANN

IBM LABORATORY VIENNA

Figure 2: Vienna Fortran


TECHNICAL REPORT

TR 25.139
20 December 1974

A FORMAL DEFINITION OF A
PL/I SUBSET

PART I

H. BEKIĆ
D. BJØRNER
W. HENHAPL
C. B. JONES
P. LUCAS

IBM LABORATORY VIENNA

TECHNICAL REPORT

TR 25.139
20 December 1974

A FORMAL DEFINITION OF A
PL/I SUBSET

PART II

H. BEKIĆ
D. BJØRNER
W. HENHAPL
C. B. JONES
P. LUCAS

IBM LABORATORY VIENNA

Figure 3: PL/1 Subset Report

However, although complex because of the liberality of its design, IBM was undeterred, as the advertisement in **Figure 4** for the language confidently demonstrates.

Reflections

Consistent with my earlier remarks about the reputation of the language, PL/1 does not loom large in IBM's own lists of amazing achievements. Company glossies celebrating their contributions make no mention of it – such as the *Think: History of Progress 1890s-2001* (IBM 2008) and the *100 Icons* (IBM 2011). However, the *IBM Journal of Research and Development* did take note of PL/1 when reflecting on IBM and high-level languages. (Sammet 1981).

Why does the language deserve so much more historical attention? One reason is that there are so many new ideas finding their way to PL/1 throughout the 1960s. Perhaps, this is not surprising if one looks at the vision and determination to pursue convergence and portability, and the state of languages at the time – as can be calibrated by the first three chapters of Sammet (1969). The number of programming constructs and design features may be overwhelming, as many commentators and users have observed. But technical ideas are there in abundance and are thought about rigorously and formally. As Radin pointed out two years after its standardisation:

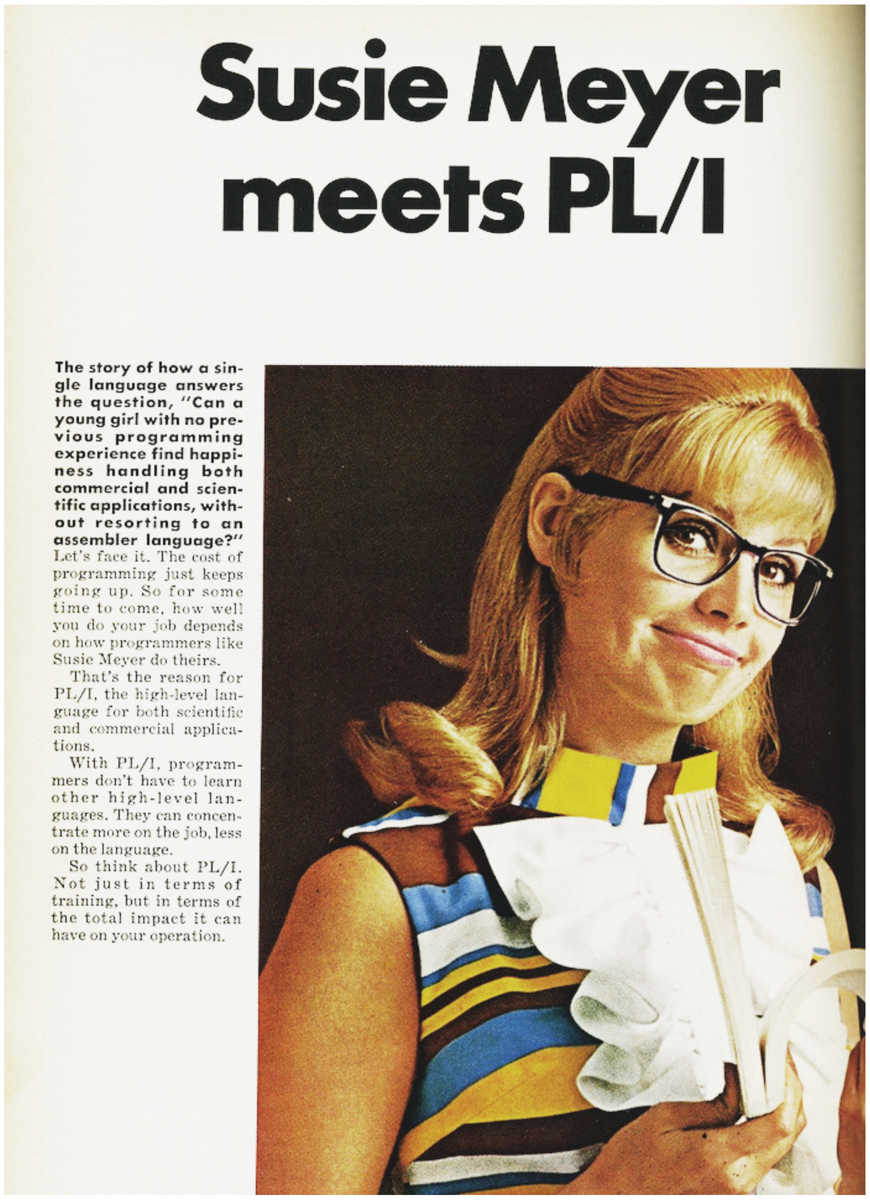
> Since PL/I took as its scope of applicability virtually all of programming, the dialogues about its various parts encompass a minor history of computer science in the middle sixties.
> (Radin 1978);

not so minor, in my opinion. Fortunately, the latest work of Cliff Jones and Troy Astarte give us new insights and incentives to rediscover PL/1 and to study the history of formal semantics of programming and programming languages. And, very fortunately, many key Vienna Lab reports, and Hursley reports, can be downloaded from Cliff Jones's library at:

http://homepages.cs.ncl.ac.uk/cliff.jones/publications/VDL-TRs

The subject need not be confined to formal methods community. For those of us of a philosophical nature, PL/1 is associated with some deep ideas and questions. Heinz Zemanek's motivations and methodological remarks in Zemanek (1966) remind us *explicitly* of the Lab's links to philosophical traditions and mathematical logic. PL/1 embodies a connection between philosophical speculations, mathematical models, formal description methods, and computing technologies active in the world. Zemanek uses the thoughts of Peirce, Russell and Wittgenstein with effect. Of course, it is important to note that the cultural foundations of the Vienna Labs owed much to the

Vienna Circle. It was Vienna was where Max Neumann learned mathematical logic and who later introduced Alan Turing to his life's work. Thus, the Vienna Circle, their precursors, fellow travellers and pupils, matter historically. Their ghosts must have



Susie Meyer meets PL/I

The story of how a single language answers the question, "Can a young girl with no previous programming experience find happiness handling both commercial and scientific applications, without resorting to an assembler language?" Let's face it. The cost of programming just keeps going up. So for some time to come, how well you do your job depends on how programmers like Susie Meyer do theirs.

That's the reason for PL/I, the high-level language for both scientific and commercial applications.

With PL/I, programmers don't have to learn other high-level languages. They can concentrate more on the job, less on the language.

So think about PL/I. Not just in terms of training, but in terms of the total impact it can have on your operation.

Figure 4: Susie Meyer

been evident in post-war Vienna.

In the formal methods community, we would do well to remind ourselves that so much of what concerns us in the digital world philosophically benefits from retracing intellectual paths that lead back to Russell and Wittgenstein.

Call for Donations

The ***History of Computing Collection*** has material on the birth and growth of semantics, but we would welcome a lot more. Dines Bjørner's important donation to the Collection is one of several concerning formal methods. Specifically, on this occasion, we do not a have anything like a full or even representative set of Vienna Lab reports nor any of the important Hursley documents. The Collection would be pleased to offer sanctuary to any materials out there in need of a safe home.

References

Troy Astarte, *Formalising Meaning: A History of Programming Language Semantics*, PhD Thesis, Newcastle University, 2019. http://homepages.cs.ncl.ac.uk/troy.astarte/res/pdf/TK_Astarte_Formalising_Meaning_2019.pdf

David Beech, A structural view of PL/I. *ACM Computing Surveys*, 2 (1) (1970), 33-64.

James W Cortada, *IBM. The Rise and Fall and Reinvention of a Global Icon*, MIT Press, 2019.

E W Dijkstra, *Concern for Correctness as a Guiding Principle for Program Composition*, EWD 288, July 1970. Department of Computer Science, University of Texas at Austen. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD288.PDF

E W Dijkstra, *A Short Introduction to the Art of Programming*, EWD316, August 1971. Department of Computer Science, University of Texas at Austen. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD316.PDF

IBM Announcement of System 360, IBM April 7, 1964. https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PR360.html

IBM 2008. *Think: History of Progress 1890s-2001* *https://www.ibm.com/ibm/history/interactive/ibm_history.pdf*

IBM 2011. *100 Icons.* https://www.ibm.com/ibm/history/ibm100/us/en/icons

Cliff Jones (editor), *Programming Languages and their Definition: Hans Bekic (1936-1982)*. Lecture Notes in Computer Science 177, Springer, 1984.

Peter Lucas, Formal semantics of programming languages: VDL, *IBM Journal of Research and Development* 25 (5) (1981), 549-561.

Peter Lucas, K. Walk, On the formal description of PL/I, *Annual Review in Automatic Programming*, Part 3, vol. 6, Pergamon Press, Oxford, 1969, 105-182.

George Radin, The early history and characteristics of PL/1, *ACM Sigplan Notices* 13 (8) (1978), 227-241.

George Radin and H Paul Rogoway, NPL: Highlights of a new programming language, *Communications of the ACM*, 8 (1) 1965, 9-17.

Jean E Sammet, *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.

Jean E Sammet, History of IBM's technical contributions to high-level programming languages. *IBM Journal of Research and Development* 25 (5) (1981), 520-534.

Ambros P Speiser, IBM Research Laboratory Zurich: The early years. *IEEE Annals of the History of Computing*, 20 (1) (1998), 15-28.

John V Tucker, The Computer Revolution and Us: Computer Science at Swansea University from the 1960s.
https://collections.swansea.ac.uk/s/swansea-2020/page/computer-science

Richard L Wexelblat (ed.), *History of Programming Languages*, Academic Press, 1981. PL/1 Session, 551559

Heinz Zemanek, Semiotics and Programming Languages. *Communications of the ACM* 9 (3) (1966), 139-143.
https://doi.org/10.1145/365230.365249

An Awkward Problem

Brian Monahan

July 2022

This all began with the following simple-sounding problem: Given a vector `vs` of non-empty sets of data elements (assume numbers), `vs` = $\langle \texttt{vs}_1, \texttt{vs}_2, \cdots, \texttt{vs}_n \rangle$, choose a (unique) element v_i from each of the sets $\texttt{vs}_i$, resulting in a *solution vector* $v = \langle v_1, v_2, \cdots, v_n \rangle$. Because of uniqueness, the solution vector v forms an *injective (one-one) mapping* from the indices of v into the overall set of elements, $\bigcup \texttt{vs}$.

Here is a more concrete example: consider the forming of a committee from a collection of *clubs* (or sub-groups), `C`, where each club `c` $\in$ `C` nominates one of their number to the committee, but in such a way that each committee member represents only one club that they belong to. Members may belong to more than one club.

To ground intuitions further, consider the following simple example – suppose the vector of sets `vs` is:

```
[
  {1, 2, 3, 4},
  {2, 5},
  {1, 3, 4},
  {2, 3, 4},
  {1, 4, 5}
]
```

Here are a couple of “solution vectors“ for `vs`having no repeated elements:

```
[ 4, 2, 1, 3, 5 ]
```

and

```
[ 1, 5, 3, 2, 4 ]
```

For the above vector of sets `vs`, there are 216 possible vectors that are *compatible* with it, where compatible here means that each ’slot’ of the vector contains some value from the corresponding set in `vs`. However, there are only 11 compatible solution vectors having no repeated elements – roughly 5

So far, so easy. In practice, the application called for solution vectors to be “random” (i.e. arbitrary) – and that, hopefully, solution vectors shouldn’t take too long to find. The input vector’s of sets would be presented arbitrarily and could be quite large – potentially having 100’s of elements. Fair enough – what could possibly go wrong? Plenty, it turned out!

1 First attempt

A fairly naive approach to this problem was taken initially, considering how trivial this all seemed to be. For example, immediate similarities could be seen with the similar task of

```
def basicAttempt(vS):
    n         = len(vS)
    vec       = vector(n, none)
    allElems  = allElements(vS)
    available = set(allElems)

    for i in {1 ··· n}:
        elems = available ∩ vS[i]
        val   = randomSelect(elems)

        vec[i]    = val
        available = available − {val}

    return vec
```

Assume that the *allElements* function computes the set D consisting of all elements occurring in the vector of sets, `vS` – and that the *randomSelect* function selects a value (pseudo-randomly) from a given *non-empty* set. Note that, for simplicity, indexing here is 1-based, rather than 0-based (see Appendix A for discussion).

The issue with this approach becomes clear when one considers the possibility of the subset `elems` computed above being empty. In that case, there is no value `val` that could be selected by the *randomSelect* function, forcing this function, and then the entire *basicAttempt* function, to fail. Therefore, the *basicAttempt* function is *not* total.

Figure 1: Naive code – *basicAttempt*

randomly choosing a permutation of a given size. This all seemed remarkably elegant and straightforward. All that was needed was to iteratively fill the slots of the selection vector using random selection of elements and keeping track of the remaining elements available.

It went something like this in Fig 1, expressed using a Python-like pseudo-code[1].

Perhaps surprisingly, this does often work for smaller examples but as `vS` gets longer and contains more objects overall, it will *eventually* fail to find any solution. Even desperately calling this *firstAttempt* code within a bounded `try - fail - retry` loop will fail to converge on a solution, no matter how many times the loop runs (see Fig 2 below).

What is happening here is that the probability of randomly choosing a candidate slot element gets closer and closer to 0, as the vector of sets, `vS`, gets longer and more and more complex. This naive approach barely works in general. To be sure, there are some exceptional special cases for which this naive approach certainly can work (such as randomly selecting a permutation or choosing from a partition where the sets are all disjoint) – but eventually, it just grinds to a halt for larger, more complex vectors `vS`, scarcely qualifying as a general method. Clearly, another approach needs to be taken!

2 Understanding SDRs and Transversals

Rather than barging in with a naive simple-minded algorithm (as done above), let's instead gain some mathematical insight into this question. To many, using mathematics might seem a

[1]See Appendix A for a fuller discussion of the pseudo-code used.

```
def repeatedAttempt(vS, limit):
    for i ∈ {1 ··· limit}:
            try
                return basicAttempt(vS)
            except
                pass

    raise ERROR
```

Figure 2: Naive code – *repeatedAttempt*

strange and perhaps pedestrian approach, but in the face of unbounded complexity, it is in fact the only possible course of action.

Readers will by now have probably realised that the problem here is akin to selecting a *transversal* for a given collection of subsets. A transversal is simply a selection of values, one from each subset, with no duplicates.

In many mathematical applications where transversals as such crop up (e.g. as with cosets in group theory), very often the subsets in question are already known to be *disjoint* – that is, forming a *partition* with no elements in common. In that case, the uniqueness condition is entirely trivial and redundant – there is always exactly one subset that each possible transversal element could belong to.

However, a more interesting situation arises where the uniqueness condition is *non-trivial* i.e. the subsets involved may overlap. In that case, instead of a transversal, one is often said to be selecting a *System of Distinct Representatives* or an SDR. From now on, we shall be considering this more general case of SDRs, rather than transversals, simply to make the uniqueness constraint clear.

By way of a recap, lets precisely define what an SDR is:

Definition: System of Distinct Representatives – SDRs

Let $\texttt{vS}$ be a vector of element sets, where the vector $\texttt{vS}$ has length $\texttt{n}$. Put $\texttt{D} = \bigcup \texttt{vS}$, the set of data elements for $\texttt{vS}$, and then let $\texttt{vec} \in \texttt{D}^{\texttt{n}}$.

$$\begin{array}{rl} \texttt{vec} \text{ is an SDR for } \texttt{vS} \Leftrightarrow & \\ \textit{Compatibility:} & \forall\, i \in \{1 \cdots n\} \bullet \texttt{vec}[i] \in \texttt{vS}[i] \\ \textit{Uniqueness:} & \forall\, i, j \in \{1 \cdots n\} \bullet \texttt{vec}[i] = \texttt{vec}[j] \Rightarrow i = j \end{array}$$

Some initial observations - an SDR is a vector of elements having exactly the same length $\texttt{n}$ as its vector of sets, $\texttt{vS}$, and therefore the data set $\texttt{D}$ of all elements needs at least $\texttt{n}$ elements for any SDR to exist.

Returning to the question of finding SDRs, it became painfully clear that the 'randomisation' approach proves not to be a fruitful avenue to explore (i.e. running out of time in more ways than one!), and at the time forced a completely different approach to be taken overall. Still, this left open a number of interesting questions for later consideration such as: Do SDRs always exist? If they don't, can they ever be found reasonably cheaply when they do exist?

2.1 Do SDRs always exist?

The simple answer to that is a definite *no*, not always. Non-existence of SDRs generally arise when there are insufficiently many unique values available to fill the slots of the SDR (i.e. a form of starvation and exhaustion). Consider the following example of 7 sets:

```
[
  {2, 6, 7},
  {2, 5, 7},
  {4, 6, 7},
  {1, 3, 5},
  {2, 4, 7},
  {4, 5, 6},
  {2, 4, 6}
]
```

Its fairly easy to see why this example doesn't have any SDRs. Firstly, note there are seven sets and only seven elements. Therefore every one of those elements must appear in any SDR. However, the two elements 1 and 3 only occur in the same set, and so any compatible vector of elements could only contain at most *one* of 1 or 3 – and never both. To fill the final slot, each of 2, 4, or 6 will have already been used at an earlier stage – and so all attempts are eventually exhausted.

3 The basic theory

There are a couple of key mathematical results to know prior to developing algorithms for SDRs. These results are all stated here without detailed proof - further details can be found in the Combinatorics literature e.g. [11, 2, 9, 13, 1, 7] . The first of these results is *Hall's Condition* for the existence of SDRs.

Theorem: Hall's Condition (Philip Hall, 1935)

Let `vS` be a vector of (non-empty) sets.

`vS` has at least one SDR $\Leftrightarrow$
For every *sub-vector* `T` of vector `vS`, $|(\bigcup \mathtt{T})| \geq |\mathtt{T}|$.

This says that SDRs exist for `vS` whenever, for every sub-vector `T` of `vS`, there are sufficiently many elements available to fill each of the slots needed for an SDR for `T`. This implies that `vS` has an SDR whenever every sub-vector also has an SDR. This is clearly a necessary condition for avoiding starvation/exhaustion. What is perhaps more surprising is that this is also a sufficient condition!

A good way to show this sufficiency is via *Ryser's Theorem*[11] which provides a numerical *lower bound* for the number of SDRs for any vector of sets `vS` satisfying Hall's Condition.

Theorem: Ryser's Theorem - Minimum number of SDRs (Herbert Ryser, 1963)

Let `vS` be a vector of (non-empty) sets of length n that happens to satisfy Hall's Condition above.

Let $k > 0$ be the least cardinality of the sets from the vector of sets, `vS`. Then there are two cases:

- If $k \leq n$, then `vS` has at least $k!$ SDRs.
- Otherwise, for $k > n$, the number of SDRs is at least:

$$\frac{k!}{(k-n)!}$$

In either case, the number of SDRs is positive, proving the sufficiency of Hall's Condition. A proof of Ryser's theorem proceeds in each case by a fairly straight-forward induction on the length of the vector of sets, vS, satisfying the Hall Condition. Naturally, if vS does satisfy Hall's Condition, then so too does any sub-vector, R, by definition.

Related to this is the observation that, for any vector of non-repeating elements, then every sub-vector it has is also non-repeating. Conversely, extending a non-repeating vector by adding a fresh, unique value results in a non-repeating vector. This property, called here the *sub-vector extension* property, will be used later when building up solution vectors algorithmicly.

Armed with these mathematical results, lets now go back to developing algorithms for finding SDRs.

4 The enumeration approach

The next algorithm moves away entirely from the probabilistic approach and instead looks at the problem from an pure;y enumeration point of view. The advantage of this approach over the previous one is its completeness of search. Although time consuming, it can be known, by enumeration, exactly how many SDRs there are for a given (smallish) system. If there is an SDR to be found, this process will find it, given enough time and resources.

Naturally enough, the enumeration approach is severely limited and can only be used in practice for small examples. However, enumeratiion is still of some use for testing purposes to provide benchmark results. The enumeration code is briefly outlined in Fig 3 below.

```
class EnumSDR():
    def __init__(self, inputVector):
        # Initialise EnumSDR using input vector.
         ...
        self.input    = tuple(inputVector)        # Input vector of sets
        self.size     = len(inputVector)      # Size of input vector
        self.position = vector(self.size, 1)      # Tracks test vector
         ...
  ...

    def enumerateSDR(self):
        results = list()
        vec = vector(self.size, none)

        for idx ∈ {1 ··· self.size}
            vec[i] = self.input[idx][1]

        # Generate vectors and test for permutation ...
        while self.moreToDo():
            if isPermutation(vec):
                results.addLast(vec)
            vec = self.advance()
        return results

    def moreToDo(self):
        # Determines when search space is exhausted
         ...

    def advance(self):
        # Calculate next test vector (in lexicographic order)
         ...
```

Figure 3: Enumeration code – *enumerateSDR*

The enumeration itself proceeds by generating each *compatible* test vector lexicographically. The enumeration completes by detecting when "rollover" occurs i.e. when the enumeration starts over again. Note that, for a given vector of sets, vS= $\langle S_1, S_2, \cdots, S_n \rangle$, then the size of the search space is equal to the total number of compatible test vectors, which is equal to:

$$|S_1| \times |S_2| \times \cdots \times |S_n|$$

5 Rethinking the search for SDRs

As we can see, SDRs may not be as common as one might have first thought. To succeed at finding such potentially rare objects, it will be necessary to rethink how the search is conducted so as to avoid being overwhelmed by a deluge of unproductive alternatives.

To be clear, all of the algorithms considered in this paper are effectively a form of search algorithm – that is, each algorithm explicitly *visits* each solution SDR, even if all the algorithm does is to count the number of solutions that were found. Naturally, some of those objects might be retained in some form.

5.1 Finding the first SDR vs. finding *all* of them?

Obviously, finding all the SDRs for a given vector of sets vS is going to potentially take far more resources to accomplish than merely looking only for one. It seems, at first sight, quite silly to consider doing anything more than is absolutely necessary.

However, the motive for asking this question at all is to reluctantly acknowledge that by attemptng to find just one SDR, then this may yet involve taking almost as much effort as looking for all of them! This may happen because, for any given input vector of sets, this may only very few SDRs, or even none at all, despite have a potentially huge solution space[2].

Moreover, it is almost certainly the case that even for algorithms finding just one SDR, they will necessarily involve *searching* in one form or another – and that this searching could then be adapted algorithmically to finding *all* of them, since it might have to explore almost the entire solution space to find the first SDR. Conversely, an algorithm for finding all SDRs can be trivially adapted to just finding the first such SDR. As a result, there is little difference in practice between algorithms that find all SDRs and those that would just find the first one – both need to perform the same *kind* of tasks.

Furthermore, it may be that a "plausibly random" choice of SDR is required rather than merely offering up the first SDR that an algorithm happens to find. A better approach might be to find the first 50 or so SDRs and then make a 'randomised' selection from this small sample. Of course, any such sampling approach is necessarily biased – but at least there is then a trade-off between the amount of bias in the selection and the amount of effort needed to finding a subset of SDRs to select from.

From now on in this paper, the algorithms given *report* in some way *all* the SDRs that are found. It is clearly trivial to adapt such an algorithm to return the first such SDR found (or indeed the 23rd SDR or even the final SDR found) if that is what is needed. In practice, rather than capture all the SDRs, an algorithm may typically capture a few, while also counting the total number found.

[2]The term "solution space" here means the entire set of compatible vectors in which the solution SDRs may be found.

5.2 A mathematical insight – isomorphic input vectors

A particular problem-reduction tactic is to consider ways to somehow preprocess the given input making it more amenable to processing, without compromising the integrity of processing and solutions found. In this section, some ways to map and rearrange input vectors are mentioned that are guaranteed to preserve number of SDRs and provide one-one correspondences with solutions. This can be useful in practice to ensure that input vectors are arranged as conveniently as possible for processing, while still able to recover the original solutions.

Let $\mathit{AllSDRs}(\mathtt{vS})$ be the set of all SDRs for the vector of sets $\mathtt{vS}$, defined as:

$$\mathit{AllSDRs}(\mathtt{vS}) = \{\ \mathtt{vec}\ \in\ D^n\ \mid\ \mathtt{vec}\ \text{is an SDR for}\ \mathtt{vS}\ \}$$

Both of the following actions result in some transformed vector of sets, $\mathtt{vS}$'.

Rearranging the input vector, vS: Let σ be a permutation of the indices of $\mathtt{vS}$. If we have that:

$$\mathtt{vS} = \langle\, S_1, S_2,\ \cdots, S_n\, \rangle$$

and then:

$$\begin{aligned} \mathtt{vS}' &= \ \langle\, S_{\sigma(1)},\ S_{\sigma(2)},\ \cdots,\ S_{\sigma(n)}\, \rangle \\ &= \ \langle\, S'_1,\ S'_2,\ \cdots,\ S'_n\, \rangle \end{aligned}$$

That is, $S'_i = S_{\sigma(i)}$, for $i\ \in\ \{1 \cdots n\}$.

Noting that since a vector is really just a mapping from indices to values, then we have, algebraically, that:

$$\mathtt{vS}' = (\sigma\ \mathrm{o}\ \mathtt{vS})$$

Renaming the data elements in vS: Let θ be a one-one mapping from D to E, where D and E are suitable sets of data elements so that $\bigcup \mathtt{vS}\ \subseteq\ D$. We can follow in a similar manner what was done above to define $\mathtt{vS}$':

$$\begin{aligned} \mathtt{vS}' &= \ \langle\, \mathit{mapSet}(\theta)(S_1),\ \mathit{mapSet}(\theta)(S_2),\ \ \cdots,\ \mathit{mapSet}(\theta)(S_n)\, \rangle \\ &= \ \langle\, S'_1,\ S'_2,\ \cdots,\ S'_n\, \rangle \end{aligned}$$

That is, $S'_i\ =\ \mathit{mapSet}(\theta)(S_i)$, for $i\ \in\ \{1 \cdots n\}$. The function *mapSet* is simply defined by the equation:

$$\mathit{mapSet}(\theta)(S) = \{\ \theta(v)\ \in\ E\ \mid\ v\ \in\ S\ \}$$

where $S \subseteq D$. Clearly, we have that $\mathit{mapSet}(\theta)(S) \subseteq E$

Noting as before that a vector is really just a mapping from indices to values, then we have, algebraically, that:

$$\mathtt{vS}' = (\mathtt{vS}\ \mathrm{o}\ (\mathit{mapSet}(\theta)))$$

In either case, it is straightforward to see that each SDR for the original vector of sets $\mathtt{vS}$naturally has a one-one correspondence to SDRs for the transformed vector of sets, $\mathtt{vS}$'. Because this correspondence is one-one between finite sets, it is clear that the number of SDRs is exactly the same for either vector of sets.

Noting that when $\mathtt{vS}$ and $\mathtt{vS}$' are vectors of sets of the same length, we say that $\mathtt{vS}$ and $\mathtt{vS}$' are **isomorphic** whenever there is a pair (σ, θ) such that $\mathtt{vS}' = (\sigma\ \mathrm{o}\ \mathtt{vS}\ \mathrm{o}\ (\mathit{mapSet}(\theta)))$ where (σ, θ) are one-one mappings of indices and data respectively.

Putting all of the above together, this shows the following theorem:

Theorem: SDR invariance property for isomorphic vectors of sets

If `vS` and `vS'` are *isomorphic* vectors of sets, then:

$$\mathit{AllSDRs}(\mathtt{vS}) \cong \mathit{AllSDRs}(\mathtt{vS'})$$

with the natural corollary that:

$$|\mathit{AllSDRs}(\mathtt{vS})| = |\mathit{AllSDRs}(\mathtt{vS'})|$$

That is, both sets have the same cardinality.

From a practical standpoint, results such as these allow developers to deploy what might be termed a 'Map/Unmap' problem reduction tactic: map the input in some convenient manner, find all the solutions for the transformed input, and finally 'unmap' those solutions back to yield solutions for the original input. Occasionally, this final 'mapping backwards' step isn't required – such as when only the number of solutions is needed, for example. This 'Map/Unmap' approach is illustrated informally by Fig 4.

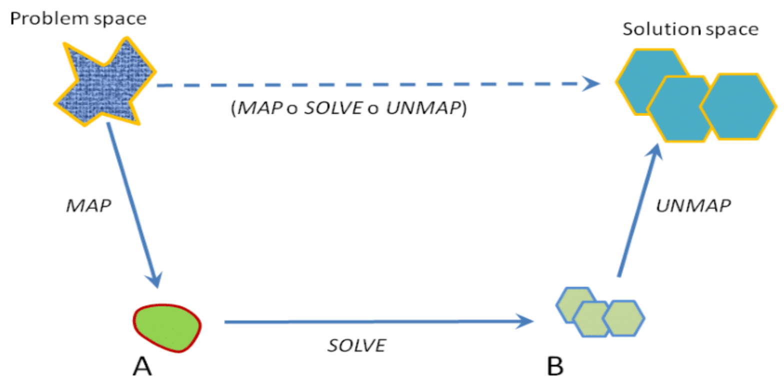


Figure 4: The (MAP o SOLVE o UNMAP) problem reduction strategy

6 A depth-first approach

Having looked quite hard at the problem now, it will probably not come as too great a surprise to see that, broadly, the solution uses some form of *depth-first search* to find SDRs[3].

To understand and gain insight into this approach, it is particularly instructive to first visualize and explore what the tree of possible solutions looks like and to see how they are arranged. To do this, consider the following basic example:

```
V = [
      {1, 2, 3, 4},
      {2, 3, 4},
      {1, 3, 4}
    ]
```

[3]For the remainder of this article, it is assumed without loss of generality that the number of candidate elements equals the length of the input vector of sets – which in turn equals the number of slots in each of the solution vectors.

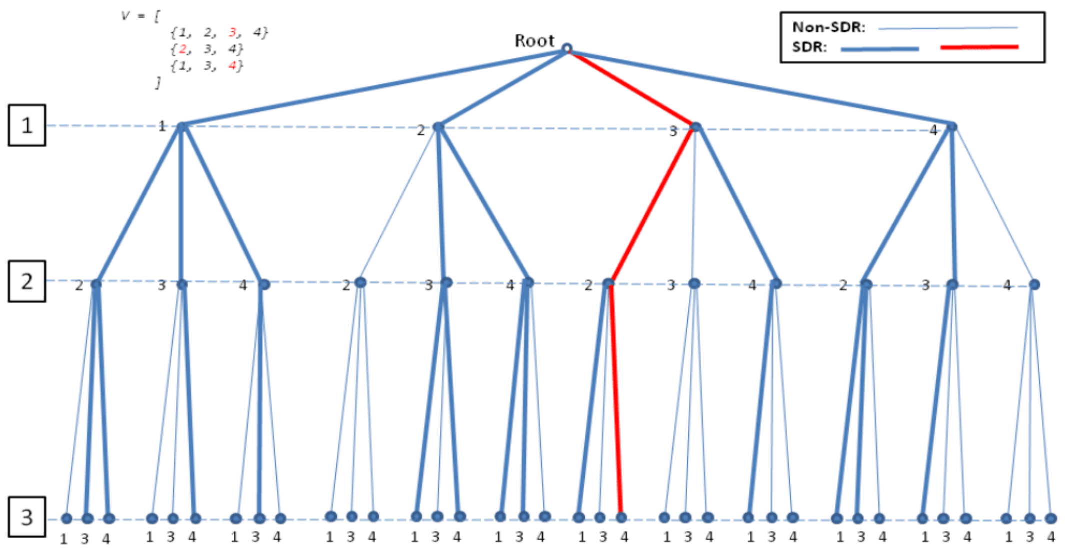


Figure 5: Solution space

Note that the indices of vector V are $\{1, 2, 3\}$, with the set of all data elements used being $\{1, 2, 3, 4\}$.

In the corresponding solution tree given by Fig 5, the indices of the vector V correspond to the *levels* of the tree. At each level, the various possible *values* that could be chosen at that level are shown at *nodes* in the tree. Thus, at level 1 (corresponding to index 1), there are four nodes correponding to the four possible values, $\{1, 2, 3, 4\}$. The solution tree has a fixed depth corresponding to the number of indices - in this case, 3.

Accordingly, each path through the solutions tree corresponds exactly to a particular solutions vector – implying that the number of endpoints is equal to the total number of solutions vectors (i.e. 36 vectors). Particular SDRs are then just those solution vectors having no repeated values – and these SDRs are denoted on the above diagram by *bold paths*, with 14 SDRs in total. The bold red path in the diagram indicates the specific SDR [3, 2, 4].

From this it becomes clear what the *depth-first* approach needs to achieve - it amounts to a dynamic traversal of the solution space, picking off the SDRs as it does so. Naturally, any efficient approach must necessarily *avoid* constructing the solution space explicitly, even though it is the very thing being explored!.

The broad idea is for the traversal to implicitly walk the tree by continually extending the current partial solution until it reaches the endpoint - if it does that successfully, then an SDR has been found. Starting from the first index, attempt to assign compatible values into the current partial solution vector, and proceed index-wise until either an SDR is found or no further progress is possible. If at any stage no further progress can be made then that part of the search is abandoned. The search continues by attempting any untried values until all possibilities have been exhausted.

The results of this traversal are handled on-the-fly by adding the SDRs to the end of a list, and are not referred to otherwise within the algorithm. This means that there are a variety of possible implementations – appending to the end of a list, outputting the SDRs on a stream for capture by an external file, or even simply counting the SDRs.

```
class FindSDRs_a():
    def __init__(self, inputVector):
        # Initialise FindSDRs using input vector.
         ...
        self.inputSets   = tuple(inputVector)          # Input vector of sets
        self.size        = len(inputVector)          # Size of input vector

        self.candidates = allElements(inputVector)
        self.curIndex   = 1

        self.currentSDR = list()

        self.results    = list()

         ...

    def main_a(self):
        self.dfSearch_a()

        return self.results

    def dfSearch_a(self):
        try:
            nextValues = list(self.inputSets[self.curIndex])
            self.exploreValues_a(nextValues)

        except NotFound:
            pass

    def exploreValues_a(self, nextValues):
        if len(nextValues) == 0:
            return

        (curValue, remainValues) = self.chooseNextValue_a(nextValues)

        if curValue is none:
            raise NotFound

        try:
            self.updateSDR_a(curValue)

            if self.curIndex > self.size:
                self.results.addTo(self.currentSDR)
            else
                self.dfSearch_a()

        except NotFound:
            pass

        # Finally, resume search for more SDRs ...
        self.exploreValues_a(remainValues)
```

Figure 6: First-cut depth-first algorithm (flawed)

```
class FindSDRs_a():
  ...

    def updateSDR_a(self, value):
        # Update the current SDR with given value and advance the index.
        self.currentSDR.addTo(value)

        self.curIndex += 1
        self.candidates.remove(value)

    def chooseNextValue_a(self, nextValues):
        # Choose next compatible value from the list of nextValues
        remainValues = list(nextValues)
        for val in nextValues:
            remainValues.remove(val)

            if val in self.candidates:
                return (val, remainValues)

        raise NotFound
```

Figure 7: The methods **self**.*updateSDR*$_a$ and **self**.*chooseNextValue*$_a$

6.1 First-cut depth-first algorithm

The broad outline of our first-cut depth-first algorithm is given in Fig 6 and Fig 7. [4].The two methods *updateSDR*$_a$ and *chooseNextValue*$_a$ update the current SDR and choose the next value from the values available respectively.

6.1.1 A significant subtlety

Overall the algorithm given in Fig 6 broadly follows the pattern outlined above – so far so good. However, an important subtlety has been negelcted which demands to be mentioned. As such, it turns out that this algorithm is *flawed*.

The algorithm has been outlined using our Python-like pseudo-code in an object-oriented manner. This means that the "state" required has been explicitly described, together with the methods that use that state to produce solutions. The bottom line here is that this gives the impression that the algorithm appears ready for implmentation. Unfortunately, that is not entirely true – if a developer were to take this algorithm literally as it stands, the implementation would surely fail.

The main issue is that the state (here called **self**) persists across *all* the recursions in a manner that fails to account for the alternatives needed to explore the solution space fully. The bulk of the work in the algorithm is done by the method *exploreValues*$_a$ which is used to recursively explore the assignment of a value for the current index [5] , **self**.`curIndex`, by appending to the current partial solution vector, **self**.`currentSDR`.

This method takes the values available for the current index and then *crucially* makes use of the currently remaining candidates to select the next value. This implies a significant dependence upon the current state, and since the current state is simply accumulated (by side-effect) throughout, it is not possible for that state to directly cater for alternative bindings and therefore further solutions. As it stands, the current algorithm does *not* use the state correctly – that is, the current algorithm is certainly *flawed*.

[4]As there are several algorithm variants discussed, significant entities are labelled with a subscript for clarity

[5]Ironically, the current index need not be represented explicity - since the next value is always appended to the current partial solution vector, and the length of this solution vector can act as a *proxy* for the current index! However, by including it explicitly, it gives an observable indicator that progress is being made.

6.1.2 Resolution

This issue may be tackled in at least a couple of ways. For example, by using a more functional pseudo-code in the first place reduces the likelihood of issues like this arising which involve an incorrect use of state. This is mostly due to functional descriptions ensuring no hidden side-effects, meaning that state information necessarily has to be managed explicitly and exactly – each use of the state becomes more trackable and accountable.

Unfortunately, this leads to a tension between familiarity and conciseness on the one hand, and evident compositional correctness on the other – with familiarity and conciseness tending to win out in practice. However, no matter what kind of pseudo-code is used, care always needs to be taken when dealing with state in the presence of recursion. With a functional pseudo-code, the appropriate care to take is easier to spot and then apply, because all the associated management needs to be considered explicitly.

In this case, note that the very last line of the $exploreValues_a$ method is a recursive call to the $exploreValues_a$ method. The requirement for this call is to *resume* the search for solutions for the current index *at entry to that method*, using any remaining values. As it stands, the algorithm incorrectly makes that recursive call using the *current state*, **self**, which will have significantly advanced since entry to the method.

Fortunately, the fix is very straightforward: make a *clone* of the state at entry to method $exploreValues_a$ using method *cloneInstance*, and then use that cloned state to initiate the final recursive call to $exploreValues_a$. Update the method $exploreValues_a$ as indicated here:

```
def exploreValues_a(self, nextValues):
    if len(nextValues) == 0:
        return

    resumeState = self.cloneInstance()

     ...

    except NotFound:
        pass

    # Finally, resume search for more SDRs ...
    resumeState.exploreValues_a(remainValues)
```

7 Further enhancement

Now that there is a working algorithm [6] at least, we can now begin to explore making this perform more efficiently (i.e. less wastefully) by finding optimisations which reduce the amount of work required to do the job. Attempting to optimize before having a working algorithm is foolish as that risks making it harder overall to complete the required task fully or even at all. Curiously enough, optimisations usually involve doing some extra preparation or perhaps using a more sophisticated representation in some manner – until the algorithm works adequately, it is best to keep it all as simple possible.

After performing a range of tests, it becomes notable that the algorithm can sometimes take a long time to complete, even when only a few SDRs are to be found. In the most extreme case, the algorithm can take a long time to process examples having *no* SDRs whatsoever! This points to the current algorithm doing all sorts of unnecessary, unproductive work that fails to yield further solutions.

[6] It is assumed that algorithms in the chosen pseudo-code are sufficiently close to an executable form that they can be realistically *tested* and practically *examined*. This is certainly the case for our Python-based pseudo-code and is also true for other modern languages as well.

Thinking this through, clearly the issue has to do with *dependent choice* which lies at the core of this entire problem – early choices of values progressively constrain later choices of value. To examine this further, consider the following example:

```
V  =  [
         {1, 2, 3, 4, 5, 6}
         {1, 2, 4}
         {1, 2}
         {1, 2}
         {1, 2, 4, 6}
         {1, 2, 4, 5, 6}
      ]
```

This has 1440 compatible vectors, but only two SDRs, [3, 4, 1, 2, 6, 5] and [3, 4, 2, 1, 6, 5] respectively.

To illustrate the effect of dependent choice, consider the following *availability matrix*, with the set of values ranging from top to bottom, and the current partial solution vector along the bottom. The check-marks (✓) indicate a possible entry value for the solution vector. At the start, the bottom row indicates a blank solution vector. All columns initially contain at least one check-mark:

1	✓	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓	✓
3	✓					
4	✓	✓			✓	✓
5	✓					✓
6	✓				✓	✓

The narrowing effect of dependent choice is illustrated in terms of the following example.

Suppose the algorithm chooses value 2 for the first position of the solution vector - this yields:

1		✓	✓	✓	✓	✓
2						
3						
4		✓			✓	✓
5						✓
6					✓	✓
	2					

Making this choice means that the column entries for the first position can be removed, and also eliminate the *remaining* entries for the value 2 - since that has now been chosen.

Looking at the second column, there are only 2 values remaining – that is, 1 and 4. Suppose the algorithm now chooses 4. Then, eliminating column and row entries, this produces:

1			✓	✓	✓	✓
2						
3						
4						
5						✓
6					✓	✓
	2	4				

Now looking at the third column, there is only one possible choice for that - the value 1. Making that *forced update*, the resulting matrix is:

1						
2						
3						
4						
5						✓
6					✓	✓
	2	4	1			

Now there is a problem - there are no values available to fill in for the fourth column - failure!

There are several conclusions to take away from this:

1. Making a choice of values constrains what is available later – choice has consequences. Once a column contains no values then no further progress can be made, forcing failure at that point.
2. Once a column has exactly one possible value in it, the value for that slot is determined and it is therefore *forced*. This forced assignment should be made as soon as reasonably possible (i.e. once it becomes *known*).
3. Any update to the solution vector may lead to a cascade of further *forced updates* – all of which should be applied. Each forced update involves *distinct* columns. However, it is also possible that more than one forced update could involve the *same* value, but in different columns. In that case, one or another of the columns will become exhausted while applying the updates – which then forces failure.
4. Depending upon the input, forced updates may occur *anywhere* within the solution vector. Supporting this desirable flexibility means the solution vector has to be implemented differently (e.g. as a dictionary).

These observations and considerations lead to some additional functionality to represent and process this matrix, together with a change in representation for the solution vector to permit updates generally. Broadly, here are the main enhancements:

Introducing a dictionary to represent the solution vector: The member, `self.currentSDR`, representing the current partial solution vector, becomes a *dictionary* mapping to allow greater flexibility when making updates at any (currently unassigned) index:

```
self.currentSDR[index] = value
```

This flexibility permits the algorithm to make forced updates as needed.

Additionally, a member, `self.vacancies`, is introduced containing the set of remaining indices to be assigned in the dictionary `self.currentSDR`. This set helps ensure that updates to the current partial solution only occur where no updates have already been made. This set starts off containing the indices $\{1 \cdots \mathbf{self}.\text{size}\}$ and reduces during the course of finding solutions.

At this point, it is worth highlighting the following important invariant:

$$len(\mathbf{self}.\texttt{vacancies}) \leq len(\mathbf{self}.\texttt{candidates})$$

Clearly, for success, this generally remains true throughout the algorithm's operation. A simple induction reveals that, assuming that this is true at entry, and noting that values and indices are used pair-wise in assignments together, this remains true throughout operation. In fact, this argument also shows that:

$$len(\mathbf{self}.\texttt{candidates}) - len(\mathbf{self}.\texttt{vacancies}) = Const \geq 0$$

where *Const* is some constant value.

Supporting the availability matrix: Because of the way this matrix would need to be updated and maintained *continually* throughout the algorithm's operation, there is a keenness to discharge these obligations as efficiently as possible. This is achieved by in fact *avoiding* a conventional representation of such a matrix (e.g. as a nested map of maps). Instead, observe that the matrix is used to anticipate exhaustion by keeping track and counting where the remaining values are. Accordingly, such functionality can be economically achieved using the following three structures:

self.indexValuePairs

This is a fixed, immutable set encoding the problem instance so that `(index, value)` pairs can be looked up. As such, it provides an *oracle* to answer queries concerning given values occurring at given indices – and does not need to be updated. This set is defined by:

```
self.indexValuePairs =
     immutable { (index, value) for index ∈ {1 ··· self.size}
                                for value ∈ self.inputSets[index] }
```

self.indexCountMap

This is an updateable mapping that keeps count of the values available at each index. This mapping is defined by:

```
self.indexCountMap =
     { index:  len(self.inputSets[index]) for index ∈ {1 ··· self.size} }
```

Finally, in the particular case where the number of candidates is equal to the number of vacancies, each candidate value necessarily occurs within every solution SDR. Since the assignment of a value to a particular index may become forced due to there only being one possible value remaining for that index, in a similar way, the assignment of a particular value at the last remaining index may also become forced. In either case, forced updates become necessary to proceed to a solution. To cater for forced updates for scarce candidates, introduce the following mapping:

self.valueCountMap

This is an updateable mapping that keeps count of the remaining occurrences for each value. This mapping is defined by:

```
self.valueCountMap =
     { value:  len(self.valueOccs(value))
          for value ∈ allElements(self.inputSets) }
```

where the method *valueOccs* calculates a set of all the remaining occurrences of a particular value in the input vector and the function *allElements* calculates the set of all value elements occurring in the input vector.

Any forced updates required at any stage can be found using the entities, **self**.indexValuePairs, **self**.indexCountMap and **self**.valueCountMap to keep track of the arrangement of values. This calculation is performed by the method **self**.*scanForForcedUpdates*. This involves looking for indices having only one remaining value and, when the number of candidates equals the number of vacancies, looking for values having one remaining index. Any forced updates that are found are applied and continues scanning until no forced updates are found. No detailed description for this method is provided here as its mostly straightforward.

With these enhancements in place, the revised algorithn is given in Fig 8.

```
class FindSDRs_b():
    def __init__(self, inputVector):
        # Initialise FindSDRs using input vector.
         ...
        self.inputSets   = tuple(inputVector)              # Input vector of sets
        self.size        = len(inputVector)              # Size of input vector

        self.candidates  = allElements(inputVector)
        self.vacancies   = set({1 ··· self.size})

        self.indexValuePairs =   ···  (see earlier discussion)
        self.indexCountMap     =   ···  (see earlier discussion)
        self.valueCountMap     =   ···  (see earlier discussion)

        self.currentSDR  = dict()

        self.results     = list()

         ...

    def main_b(self):
        self.dfSearch_b()
        return self.results

    def dfSearch_b(self):
        try:
            nextIndex = self.chooseNextIndex_b()
            nextValues = list(self.inputSets[self.nextIndex])

            self.exploreValues_b(nextIndex, nextValues)

        except NotFound:
            pass

    def exploreValues_b(self, curIndex, curValues):
        if len(curValues) == 0:
            return

        resumeState = self.cloneInstance()

        (curValue, remainValues) = self.chooseNextValue_b(curValues)

        try:
            self.updateSDR_b(curIndex, curValue)
            self.scanForForcedUpdates()                     # Scan and apply any forced updates

            if:
                self.results.addTo(self.currentSDR)
            else:
                self.dfSearch_b()

        except NotFound:
            pass

        # Finally, resume search for more SDRs ...
        resumeState.exploreValues_b(curIndex, remainValues)
```

Figure 8: Revised depth-first algorithm

```
class FindSDRs_b():
    ...

    def updateSDR_b(self, index, value):
        # Update the current SDR at the given index and value.
        self.currentSDR[index] = value

        self.vacancies.remove(index)
        self.candidates.remove(value)

        # Update indexCountMap and valueCountMap mappings
            ...
            ...

    def chooseNextIndex_b(self):
        try:
            return min(self.vacancies)
        except:
            raise NOTFOUND

    def chooseNextValue_b(self, nextValues):
        # As for hooseNextValue_a
         ...
```

Figure 9: The methods **self**.*updateSDR*$_b$, **self**.*chooseNextIndex*$_b$ and **self**.*chooseNextValue*$_b$

7.1 A short, informal argument for correctness

An informal correctness argument for the algorithm in Fig 8 goes as follows. This is essentially an inductive argument based upon the depth and therefore the size of the solution space. This process is guaranteed to terminate since the depth of recursion is limited by the number of vacancies available.

The algorithm primarily conducts a depth-first traversal search over the tree of possible solution vectors, as illustrated for example by Fig 5. This search is mainly the focus of the *exploreValues*$_b$ method. The search proceeds index-wise, and successively chooses the next value, if possible, from the corresponding range of values for the current index.

The current partial solution is extended when a value is found that is compatible with the index and the currently available candidates, ensuring that the extension also satisfies the uniqueness constraint for SDRs (c.f. the *sub-vector extension* property mentioned earlier). If this choice of index and value then completes the current partial solution vector, then an SDR has been found and can be reported. Otherwise, there is further work to do and a recursive call of *dfSearch*$_b$ is made to capture all remaining solution SDRs that *consistently* extend the current partial solution vector so far. To ensure completeness of search, the final step of *exploreValues*$_b$ explores any remaining values for the current index by using a further recursive call to *exploreValues*$_b$, taking care to resume from the state at *entry* to the method.

From this, it can be seen that the method *dfSearch*$_b$ proceeds by gathering solutions from deep within the search space, going 'vertically' from the current partial solution towards the tips, whereas the method *exploreValues*$_b$ proceeds 'sideways' by exploring alternative solutions 'horizontally across'. In this way, the solution space is covered, as schematically diagrammed in Fig 10.

This algorithm incorporates the use of forced updates. Critically, such updates are those that are forced with respect to the current partial solution vector – and as such, would have eventually been made in every extension from the current partial solution vector that produces SDRs. Using forced updates adds constraints to eliminate extensions that are *not* consistent with the current partial solution vector and accelerates finding those extensions that are consistent with

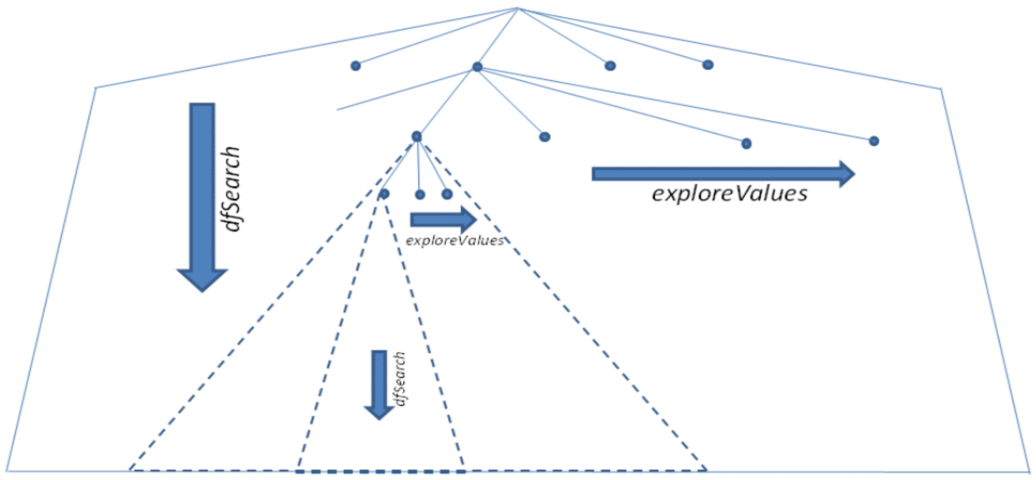


Figure 10: Recursive traversal of solution space

the current partial solution vector. As such, the use of forced updates does not affect completeness of search since using forced updates only preserves solutions and never removes them.

7.2 Does the algorithm work in practice? And how well does it do it?

The algorithm given in Fig 8 has been informally tested against a battery of 30+ examples and compared against the results of performing an exhaustive (and exhausting!) pure enumeration of SDRs. Because some of these examples would yield a large quantity of solutions, the number of SDRs captured per run was limited to 40. The examples range from the utterly trivial to sizable examples involving up to 13 elements and having solution spaces with a size in excess of 10^10 vectors. In all cases, both the enumeration and the revised algorithm produce the same overall counts of SDRs found, providing some empirical validation.

However, there is a radical difference between the times taken by enumeration and the times taken by the recursive depth-first search algorithms given here. Of course, this is entirely to be expected – the enumeration approach visits every completed vector in solution space and the run-time is therefore proportional to the size of the solution space. The depth-first search algorithms efficiently prune the search as soon as exhaustion occurs at any level, since the current partial solution vector cannot then be consistently extended into an SDR. This pruning has a radical effect on run-time – it prevents much fruitless searching which attempts to extend inconsistent solution vectors.

Furthermore, by adding the use of forced updates, this measure can curtail unproductive searches for solutions even quicker, sometimes radically so. The trade-off being made is between the amount of work to locate and apply forced updates at each node visited, and the work needed to visit every node obliviously, regardless of potential success.

Generally, the trade-off empirically appears to be very much in favour of detecting and applying forced updates, particular for larger examples.

7.3 Adding priority

Earlier, the idea that mapping and rearranging the input vectors of sets might prove advantageous was suggested (see discussion near Fig 4). For the pure enumeration approach, mapping and rearrangement the input makes very little difference to runtime. Technically, all that changes is the particular order in which vectors are enumerated, and therefore visited.

However, as one might expect, for the case of the recursive depth-first algorithms, the mapping and rearranging of the input vectors[7] often does make a considerable and radical difference to the run-time, as is easily confirmed empirically for larger examples.

What is going on here? The key point is that the overall approach involves either extending the current partial solution when feasible or otherwise rapidly abandoning failed and therefore unproductive alternatives. Now, the reason that arranging the input vector can matter is that there may be elements having smaller numbers of occurrences than others. Those elements having fewer number of occurrences are, in some sense, more likely to cause starvation and exhaustion. These scarce and rarely occurring elements therefore tend to need *critical placement* within the current partial solution that can then be extended to SDRs.

This means that by considering such scarce elements sooner, forced updates can more effectively either accelerate the finding of SDRs because critical elements are put in place sooner – or alternatively, help hasten the abandonment of unproductive searching due to exhaustion of alternatives.

The overall conclusion is that, for the recursive depth-first algorithms, performance can be radically enhanced for many input vectors by initially sorting and rearranging the input vectors so that any scarce elements are considered as soon as possible during algorithm operation. This step, called *prioritisation*, is an efficient precursor to the main processing and in effect *canonicalises* the input so that isomorphic input vectors are brought to the same form.

Given that this step is effectively a form of precomputation, this does not materially affect the structure and content of the algorithms already presented. This step can therefore be assumed to have been applied prior to algorithm operation.

7.4 Some empirical outcomes?

Broadly, the effect of these enhancements is radical, particularly for larger test examples that have relatively few (or even no) SDRs to be found.

To put the relative efforts expended by each algorithm into perspective, the pure enumeration approach took around 28 hours 7 minutes to tackle the 30+ examples mentioned earlier, whereas the extended recursive depth-first algorithm (with forced updates and prioritisation in force) took around 2 minutes 40 seconds total run-time when applied to the same test examples. Bearing in mind that this overall runtime is for searching for *all* SDRs (not just finding the first one) then this does seem quite impressive[8].

None of these algorithms made extensive demands upon memory - and all were implemented in Python and run on the same computer.

Given that the test set is not open to scrutiny, statements like the above are not subject to critical review. To address these criticisms, the following website will contain open source code for the algorithms, test harness, documentation, and examples:

`https://github.com/BrianMonahan/SDR`

[7]From now on, assume that a vector of lists representation is used, instead of vectors of sets, so that an ordering upon elements may be used.

[8]Even so, one should be extremely cautious of empirical results of this kind – only a small number of unsystematic examples have been tried out. In the absence of any argument for efficiency, its reasonable to expect that 'bad' examples will be found having poor efficiency for their size.

8 Related work

It turns out that there is a considerable history to the general problem of finding SDRs – but where this topic arises within *Graph Theory*[14] under the guise of finding *matchings* within *bipartite graphs*.

To recap, a bipartite graph is an undirected graph (network) whose set of vertices is split into two non-empty disjoint sets so that each edge of the graph has an endpoint in each set (i.e. no loops). In this context, a *matching* is then a subset of edges that shares no endpoints i.e. all the edges are disjoint from one another. A *maximal matching* is then a matching where no further edges could be validly added to *extend* the matching. Note that there could be many different maximal matchings for a given bipartite graph.

Now, SDRs generally correspond to maximal matchings within a certain bipartite graph where the set of indices corresponds to one set of nodes and the set of data elements corresponds to the other set of nodes. The graph itself is then formed by linking those nodes representing indices to the nodes representing values for each corresponding index (as specified by the input vector). Additionally, the matchings representing SDRs are naturally expected to cover each of the nodes corresponding to the indices.

The problem of finding maximal matchings has been long studied within Graph Theory. One of the early algorithms for finding a maximal matching is due to the work of Edmonds[4], as discussed in Chapter 5 of [6]. The ideas of *alternating* and *augmenting paths* are introduced that zig-zag between the two sets of nodes and do not self intersect. Matchings can then be extracted from these paths, by taking *alternate* edges along any given path as the edges of the matching, yielding two matchings, one longer than the other. The remaining challenge lies in systematically exploring the augmenting paths needed for maximal matchings, leading to algorithms with complexity $O(EV)$, where E is the number of edges and V is the total number of vertices.

A significant step forward was the development in 1973 of the Hopcroft-Karp algorithm for maximal matchings in bipartite graphs[8], and is one of the most efficient algorithms known for finding maximal matchings. This makes use of the augmenting paths ideas mentioned above, and uses *breadth-first search* to extend the set of augmenting paths to a maximal matching – the total run-time of this algorithm to find a maximal matching is $O(E\sqrt{V})$, a considerable improvement.

The online lecture notes from Clifford Stein on this topic provides a useful introduction and summary of this area[12]. A very recent in-depth exploration of algorithmic work in this area can be found in Chapter 25 of the 4th edition of the classic textbook[3].

Finally, the problem of finding maximal matchings can also be treated as a network-flow problem and solved using techniques such as the Ford-Fulkerson algorithm[5].

9 Some Observations

This paper has explored some algorithms for finding SDRs – this has involved a couple of refinements that has been generally led throughout by mathematically expressed considerations. The algorithms developed dealt with finding all SDRs, rather than just one, as explained in Section 5.1. An informal argument was given in Section 7.1 for the correctness of the algorithm. Some informal empirical evidence was stated for both correctness and overall efficiency in Section 7.4.

All that being said, the overall account suggests there remains something of further interest and study, despite the related work reported in Section 8. The work here is at least suggestive of questions such as:

- Giving a more formal argument for correctness.
- Giving a worst-case/average-case complexity analysis.

Given that the time taken to find all SDRs is dependent upon the number of SDRs there are to be found (since each SDR would need to be visited), is there much to be said about the time taken finding the first SDR relative to the time taken finding all of them? Clearly, the former is shorter than the latter. Specifically, what is the relationship between these times – how linear is that relationship for example? In part, the subtlety here concerns how well does the algorithm do in avoiding non-solutions – can that be quantified in some meaningful way?

Even if the focus on finding *all* SDRs seems hopelessly academic and deeply wasteful – *why on earth would anyone need to do that?* – it nonetheless provides a challenge of the trade off between finding some solution against finding them all (or at least, counting how many there are). The related work outlined in Section 8 focuses on complete algorithms for maximal matchings – how well do these algorithms generalise to the broader question of finding/counting all maximal matchings?

9.1 Requirements, Algorithms and Programs

Ideally, the conventional mantra in Formal Methods seems to go something like this: Firstly, formulate your requirement in some mathematically respectable manner, within a formal notation. Secondly, perform a number of elegant refinements to reach a polished, executable program. Of course, the analogy being appealed to here is with proving a mathematical theorem by using a set of inference rules in a formal logic. However, instead of inference rules as such, program proofs make use of refinement and reification steps at a range of granularities, from big to small. The activities of design derivation and verification is all part and parcel of the same unified process.

What is advocated here instead takes a more humble, hopefully realistic and perhaps less aggressive approach than the parody stated above. For example, it is rather hard to formally prove a particular mathematical theorem first off just given a set of inference rules, even before it has been informally understood and stated. Formally credible proofs of mathematical results are of course worth while and necessary – but they tend to arrive towards the *end* of the process, and only very rarely at the beginning – certainly true for non-trivial results. The erection of a mathematical edifice to characterise an entire theory (one having mathematical depth, such as Combinatorics, for example) is like building a large house – it needs architectural insight and overall planning by many people for this to work well and to place all the primary results in their proper position and relationship to one another (Feng Shui for mathematics, perhaps?). The same holds also when developing and providing software applications and libraries, as well as for other areas of engineering – all need a sense of the components working effectively in combination with each other - which requires some archetectural sensibility.

Instead, the common discovery and design patterns already practiced widely in mathematics and engineering could be adopted. These start off with finding an approximation of some kind (often intuitive) and then continue by refining and honing, sometimes with complete restarts when the trail goes cold and somewhere else looks more promising, until finally workable solutions in the form of algorithms are reached which may lead to useful programs. It's messy, but it can work.

Here I began with a question, a requirement. In practice, it is never as clear cut as that - but lets accept that there is a requirement. I then leapt to a particularly naive algorithm which, although it could work, it couldn't at scale, I went back to basics with a pure enumeration algorithm that I knew could go some of the way towards a solution, but much more would be needed. There was then a good deal of mathematical reflection, reading and research, until I realised that good old fashioned depth-first search might do the trick. The big step forwards was to realise that forced updates could radically cull non-productive alternatives. followed by an observation that systematically rearranging the input could have significant benefits.

What is an algorithm? An algorithm is itself rather like an abstract program, structurally characterising some behaviour to achieve a desired goal, for some given input. In the classic textbook[3], an algorithm is "any well-defined computational procedure that takes some value or set of values as **input** and produces some value or set of values as **output**.". Algorithms can be expressed in different ways within different pseudo-codes and yet be classed as representing the same 'computational procedure' (c.f. Turing machines, Lambda calculus).

For me, algorithms are *mathematical* entities, just like numbers and functions are, if albeit rather complicated entities. Algorithms have properties and can be reasoned about with logic. Algorithms can be compared and contrasted. Algorithms can achieve certain goals – or fail to achieve them; determining which alternative is the case may often be highly non-trivial to know. Finally, algorithms can be implemented by concrete programs – or not. Understanding all of this in practice forms a part of Formal Methods and more widely, Computer Science.

A Remarks on the Python-like pseudo-code

Given that there is quite a lot of pseudo-code given in this paper, ostensibly for a formal Computer Science audience, I feel somewhat duty-bound to discuss at least some of the ins and outs of the pseudo-code notation used. Although the details are really only indicative at best, they are hopefully sufficiently useful to understand the algorithms.

The idea of using a pseudo-code is to remove enough detail to allow the content to be compactly described, and yet convey sufficient information for a skilled developer to construct their own program following the described algorithm. There is clearly a tension in using pseudo-code to describe an algorithm - is it drowning in too much unnecessary detail or is there not enough information to be useful and helpful?

By its very nature, pseudo-code is therefore sketchy and incomplete, an inevitable compromise. Giving a formal semantics to something as sketchy as a pseudo-code seems a faintly ridiculous and pointless activity. Instead, a brief outline is given of the pseudo-code so that readers can better grasp what the content is. Particular attention is given to some areas where subtleties are known to lurk, so that these can be highlighted and given some explanation for readers.

This pseudo-code is very broadly similar in syntax and intended meaning to Python[10], having keywords, forms and structures that should be familiar to many developers; Python was chosen here because of its broad and sustained popularity among developers, as well as its relative simplicity and economy of expression. As with Python, *indentation* is used here to indicate nested block structure, in place of the semi-colons and braces ({, }) used in C-like notations.

A.1 Datatypes: Lists, Sets and Dictionaries

The notation here is generally routine, but with the following general exceptions to how Python handles them.

Indexing: As remarked earlier, lists and so on are indexed from 1 upwards i.e. `lst`[1] stands for the *first* element of list `lst`. This simplifies indexing a little for describing an algorithm – for example, indexing runs from 1 to `n` rather than from 0 to (`n`-1). From a programming point of view, this means that indexing in this document would translation from a 1-based indexing to a 0-based indexing for practical programming purposes.

Lists, Tuples and Vectors: These are each very similar to their Python namesakes. Lists are mutable sequences – they can both modified and added to; tuples are similar to lists, except they are immutable and cannot be modified or added to. Vectors are the same as tuples except they are defined to have a certain fixed size (i.e. cannot be added to) but each “slot” or “field” can be modified/updated. All of these structures are indexed from 1.

Sets: These are not quite the same as sets in Python. Here, sets are considered to be *ordered lists (least to greatest), without repetition* – that is, adding the same element twice does nothing. In particular, this means sets can be indexed just like lists can (i.e. something that Python does not allow). Moreover, the first element is always the least element of the set, with the last element of the set always being the greatest, and so on.

Dictionaries (or Maps): These are generally are the same as dictionary mappings in Python. They are regarded as having efficient lookup etc. and are mutable/updateable. Here is a literal dictionary of five elements:

```
{1: 2, 2: 53, 5: 26, 7: 50, 8:457 }
```

Immutability: Structured values like lists, sets and dictionaries can be marked as *immutable*[8] to indicate that their value cannot be modified – that is, once defined, its value remains fixed and constant. For example, the list:

```
listOfSquares = immutable [ 1, 4, 9, 25, 36, 49, 64 ]
```

would be equivalent to a tuple as it has been marked as immutable, preventing subsequent modification.

A.2 Some other notation

The notation *len*(obj) is overloaded to generally stand for the *size* of an object `obj` – that is, for the *cardinality* of a set, for the *length* of a list, tuple or vector, as appropriate, or for the number of entries in a dictionary.

`class MyClassName():`
`    def __init__(self, <arg>, ··· , <arg>):`
`        # Constructor function for class`
`        self.member1 = <expression>`
`        ···`
`    # Method definitions`
`    def myMethod1(self, <arg>, ··· , <arg>):`
`        ···`
`    def myMethod2(self, <arg>, ··· , <arg>):`
`        ···`
	Class definitions, with constructor function and methods. The special parameter **self** provides an explicit class instance reference.

`def myFunction(<arg>, ··· , <arg>):`
`    <statement>`
`    ···`
`    <statement>`
	Function/method definition

`if <condition>:`
`    <then-code-block>`
`elif <condition>:`
`    <elif-code-block>`
`else:`
`    <else-code-block>`
	If-style conditional statement.

`try:`
`    <code-block>`
`except MyException:`
`    <code-block>`
`except:`
`    <code-block>`
	Try-Except exception handling statement.

myLst $=$ $[(x-7)^2+5 \quad \textbf{for } x \in \{3 \cdots 37\} \textbf{ if } x\%3=2]$ mySet $=$ $\{(x-7)^2+5 \quad \textbf{for } x \in \{3 \cdots 37\} \textbf{ if } x\%3=2\}$ myMap $=$ $\{x:(x-7)^2+5 \quad \textbf{for } x \in \{3 \cdots 37\} \textbf{ if } x\%3=2\}$	List, Set and Dictionary/Map Comprehensions

Figure 11: Some pseudo-code explained

[8]Python supports immutability for certain datatypes (e.g. `frozenset`, `tuple`) but not uniformly.

References

[1] Victor Bryant *Aspects of Combinatorics - A wide ranging introduction* CUP, 1992

[2] Peter J. Cameron *Combinatorics: Topics, Techniques, Algorithms* CUP, 1994

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 4th ed., MIT, 2022

[4] J. Edmonds *Paths, trees and flowers*, Can. J. Math., Vol 17, 449-67, 1965

[5] L. R. Ford, D. R. Fulkerson *Flows in networks*, Princeton University Press, Princeton, NJ 1962

[6] Alan Gibbons *Algorithmic Graph Theory*, CUP, 1985

[7] John M. Harris, Jeffry L. Hirst, Michael J. Mossinghoff *Combinatorics and Graph Theory* Springer, 2000

[8] John E. Hopcroft, Richard M. Karp *An* $n^{5/2}$ *Algorithm For Maximum Matchings In Bipartite Graphs* SIAM J. of Computing, Vol 2, No. 4, December 1973

[9] J. H. van Lint and R. M. Wilson *A course in Combinatorics*, CUP, 1992

[10] Python, `https://www.python.org/`

[11] Herbert John Ryser *Combinatorial Mathematics* The Carus Mathematical Monographs, No. 14, MAA, 1963

[12] Clifford Stein *Online lecture notes for IEOR 8100 Matchings - Lectures 4, 5, and 6*, `http://www.columbia.edu/~cs2035/courses/ieor8100.F12/index.html`, 2012 (*Try browsing for:* `Cliff Stein course matchings 2012`)

[13] Douglas B. West *Combinatorial Mathematics* CUP, 2021

[14] Robin J. Wilson *Introduction to Graph Theory*, 5th ed., Pearson, 2010

Ib Holm Sørensen: Ten Years After

Jonathan Bowen
July 2022

It is ten years since Ib Holm Sørensen, that rare breed of both a formal methods researcher and practitioner, passed away at the early age of 62, in the centenary year of the birth of the computer science pioneer Alan Turing. This article considers Ib Sørensen's life and work, especially regarding his contribution to the field of formal methods. In 1981 he achieved his DPhil at the Programming Research Group in Oxford under Tony Hoare [1] and his further contributions there helped lead to a Queen's Award for Technological Achievement for the IBM CICS Project in 1992 [2,3].

Ib Sørensen (29 January 1949 – 17 January 2012) was a computer scientist who made important contributions to the early development and application of formal methods, especially the Z notation and B-Method, working in both academia and industry [4]. Born in Aabenraa, Denmark, Ib Sørensen started his academic career in the 1970s at Aarhus University, where he worked on the Rikke-Mathilda microassemblers and simulators running on the DECSystem-10 computer [5].

In 1979, Ib Sørensen joined the Programming Research Group, part of the Oxford University Computing Laboratory (now the Oxford University Department of Computer Science) in England, under the leadership of Prof. Tony Hoare. There he worked with Jean-Raymond Abrial, Bernard Sufrin, and others, making contributions to the early development of the formal specification language Z. He gained a DPhil degree from the University of Oxford in 1981, with Tony Hoare as his advisor [1] (see also appendix for some extracts). He taught early courses on the Z notation at Oxford [6] and established the Z User Meeting series there in 1985, which continues as the ABZ international conference combined with other state-based formal methods including ASM and the B-Method to this day.

Ib Sørensen led the Transaction Processing Project at Oxford from its inception in 1982 (later the "CICS Project" [7]), collaborating with IBM (UK) Laboratories [8]. The project formally specified parts of IBM's CICS transaction processing software using the Z notation. This won a Queen's Award for Technological Achievement in 1992 [2,3]. As part of the CICS Project, Ib Sørensen extended the Guarded Command Language of Edsger W. Dijkstra using the Z schema notation as abstract commands [9]. These ideas were later formalized by Carroll Morgan in his refinement calculus [10]. Ib Sørensen was also a co-author of the seminal *Specification Case Studies* book on the use of Z, first published in 1987 (second edition in 1993) [11].

From the late 1980s, Ib Sørensen was central in the development of the B-Method, a leading formal method [12]. He left Oxford University to lead a team at BP Research [13], developing the B-Tool to provide tool support for the B approach. He then founded the company B-Core (UK) Limited to support the B-Toolkit [4,14], a set of programming tools designed to support the use of the B-Tool, and to undertake B-related projects. Ib Sørensen's help and advice have been acknowledged in textbooks on the B-Method [14,15].

Latterly, Ib Sørensen returned to the University of Oxford. From 1999, he worked on the B-based *Booster* models of requirements. He died of a stroke early in 2012 while in Fort-de-France, the capital of Martinique in the Caribbean, before he was able to retire [4].

Ib Sørensen was a "doer" and as such his publications do not reflect his contribution to the field of formal methods in an adequate way. Unusually, he resigned his academic post at Oxford, normally a lifetime position for most at the university once they have achieved it, to join industry, first at BP, and then at his own company B-Core. With his foundational and practical contributions to both the Z notation and the B-Method, he has been an important figure in the formal methods community. As a person, he was kind and thoughtful, always understated in his interaction with colleagues. His modesty has perhaps meant that his contribution to formal methods has been underappreciated. This brief tribute aims to redress that in a small way.

Selected publications

Ib Sørensen co-authored the following [16,17,18].

At Aarhus University:

- Ib Holm Sørensen, Eric Kressel (1975). *A proposal for a multi-programming BCPL system on RIKKE-1* (in Danish). Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Eric Kressel, Ib Holm Sørensen (1975). *The first BCPL system on RIKKE-1*. Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Eric Kressel, Ib Holm Sørensen (1975). *The Mathilda driver, a software tool for hardware testing* (in Danish). Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Ib Holm Sørensen, Eric Kressel (1977). *RIKKE-MATHILDA microassemblers and simulators on the DECsystem 10*. DAIMI Report Series, MD-28. Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Ib Holm Sørensen (1978). *System Modelling: a Methodology for Describing the Structure of Complex Software, Firmware and Hardware Systems Consisting of*

Independent Process Components. DAIMI Report Series, PB-87. Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.

- Jens Kristian Kjærgård, Ib Holm Sørensen (1980). *BCPL on RIKKE* (in Danish). DAIMI Report Series, MD-36. Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Jens Kristian Kjærgård, Ib Holm Sørensen (1980). *The RIKKE Editor* (in Danish). DAIMI Report Series, MD-37. Matematisk Institut. Datalogisk Afdeling, Aarhus University, Denmark.
- Ib Holm Sørensen (September 1981). *Specification and Design of Distributed Systems*. DAIMI Report Series, PB-141. Aarhus University, Denmark. doi:10.7146/dpb.v10i141.7416

At Oxford University:

- Ib Holm Sørensen (September 1981). *Topics in Programme Specification and Design: Specification and Design of Distributed Systems*. DPhil thesis. Wolfson College, University of Oxford, UK.
- Bill Flinn, Ib Holm Sørensen (January 1986). "CAVIAR: A Case Study in Specification". In Tosiyasu L. Kunii (ed.), *Application Development Systems*. Springer, pp. 126–164. doi:10.1007/978-4-431-68051-2_8
- C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, B. A. Sufrin (August 1987). "Laws of programming". *Communications of the ACM*. **30** (8): 672–686. doi:10.1145/27651.27653
- Steve King, Ib Holm Sørensen, J. C. P. Woodcock (July 1988). *Z: Grammar and Concrete and Abstract Syntaxes (Version 2.0)*. Technical Monograph PRG-68. Programming Research Group, Oxford University Computing Laboratory, UK.
- J.-R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, I. H. Sørensen (1991). "The B-method". In S. Prehn, H. Toetenel (eds.), *VDM '91 Formal Software Development Methods*. Springer, Lecture Notes in Computer Science, Vol. 552, pp. 398–405. doi:10.1007/BFb0020001
- Bill Flynn, Roger Gimon, Steve King, Carroll Morgan, Ib Holm Sørensen, Bernard Sufrin (1993). In Ian Hayes (ed.), *Specification Case Studies*. (2nd ed.). Prentice Hall International Series in Computer Science. ISBN 978-0-13-832544-2. (1st ed., 1987.)
- Dave S. Neilson, Ib Holm Sørensen (October 1994). "The B-Technologies: a system for computer aided programming". In *6th Nordic Workshop on Programming Theory*. BRICS.
- Jim Davies, Charles Crichton, Edward Crichton, David Neilson, Ib Holm Sørensen (2005). "Formality, evolution, and model-driven software engineering". *Electronic*

Notes in Theoretical Computer Science, **130**: 39–55. doi:10.1016/j.entcs.2005.03.004

References

1 Sørensen, Ib Holm (September 1981). *Topics in Programme Specification and Design: Specification and Design of Distributed Systems* (DPhil thesis). Wolfson College, University of Oxford, UK.

2 King, Steve (1993). "The Use of Z in the Restructure of IBM CICS". In Hayes, Ian (ed.), *Specification Case Studies* (2nd ed.), pp. 202–213. Prentice Hall International Series in Computer Science.

3 "Prof. Jim Woodcock, FREng". University of York, UK.

4 Roscoe, Bill (8 February 2012). "Ib Sorensen – In memoriam". Department of Computer Science, University of Oxford, UK.

5 Sørensen, Ib Holm; Kresse, Eric (December 1977). *RIKKE-MATHILDA microassemblers and simulators on the DECSystem-10*. Technical report DAIMI MD-28. Aarhus University, Denmark.

6 Woodcock, Jim; Davies, Jim (1996). "Acknowledgments". *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. ISBN 978-0139484728.

7 Fitzgerald, J. S. (October 2006). *Perspectives on Formal Methods in the Last 25 years*. Technical Report Series, CS-TR-983. Newcastle University, UK.

8 Hayes, Ian (1993). "Preface to the first edition". *Specification Case Studies* (2nd ed.). Prentice Hall International Series in Computer Science.

9 Hayes, Ian J.; King, Steve (2021). "Industry Influence on Research". In Jones, Cliff B.; Misra, Jayadev (eds.), *Theories of Programming: The Life and Works of Tony Hoare*, section 11.9, pp. 266–267. Association for Computing Machinery. ISBN 978-1450387286.

10 Morgan, Carroll (1994). *Programming from Specifications* (2nd ed.). Prentice Hall International Series in Computer Science. ISBN 978-0131232747.

11 Hayes, Ian, ed. (1993). *Specification Case Studies* (2nd ed.). Prentice Hall International Series in Computer Science. ISBN 978-0-13-832544-2.

12 Bhattacharya, Sourav; Winter, Victor L., eds. (2012). "History of B". In *High Integrity Software*, section 8, p. 40. Springer. ISBN 978-1461513919.

13 Crichton, Edward (29 March 2022). "BToolkit". GitHub.

14 Wordsworth, John B. (1996). *Software Engineering with B*. Addison-Wesley. ISBN 978-0201403565. (Inside cover.)

15 Schneider, Steve (2001). *The B-Method: An Introduction*. Palgrave, Cornerstones of Computing. ISBN 978-0333792841. (Acknowledgements.)

16 "Ib Holm Sorensen". Google Books.

17 "Ib Holm Sorensen". Google Scholar.

18 "Ib Holm Sorensen". WorldCat.

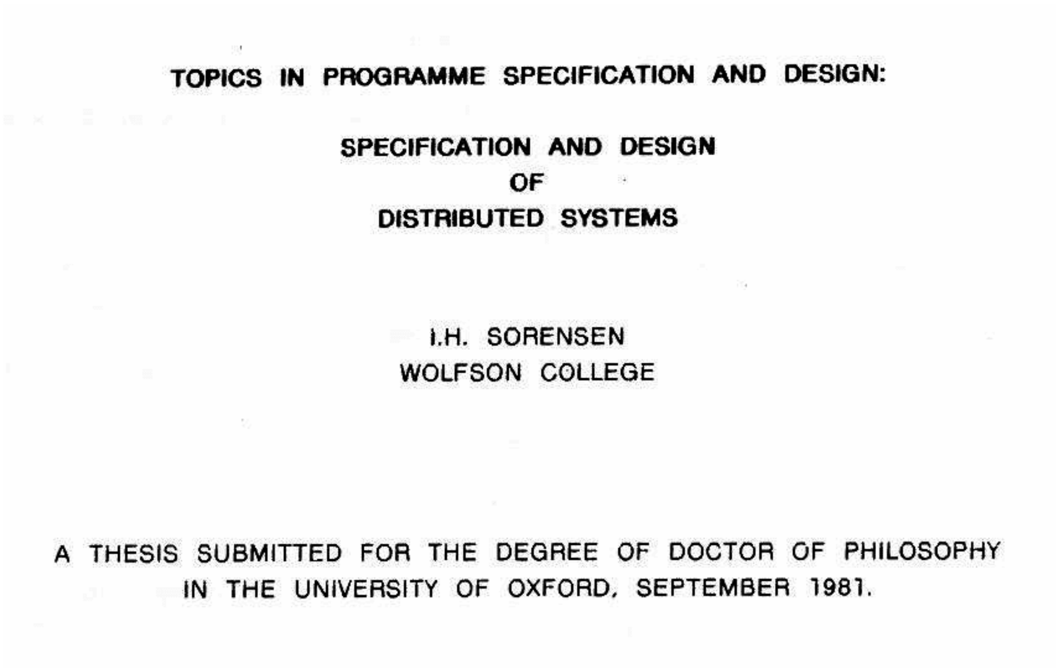
Further links

- Personal home page (http://www.cs.ox.ac.uk/people/ib.sorensen/). University of Oxford, UK.
- Ib Holm Sørensen (https://dblp.org/pid/91/4074). DBLP Bibliography Server.
- Ib Sorensen (https://www.linkedin.com/in/ib-sorensen-a16289b/). LinkedIn.
- Ib Holm Sørensen's scientific contributions (https://www.researchgate.net/scientific-contributions/Ib-Holm-Sorensen-69788930). ResearchGate.

Appendix: Extracts from DPhil thesis

Below are some extracts from Ib Sørensen's 1981 doctoral thesis [1], supervised by Tony Hoare at the Programming Research Group in Oxford, and using an early version of the Z notation. Jean-Raymond Abrial was based at the PRG at this time, developing the Z notation. Bernard Sufrin was also using the Z notation and Cliff Jones was studying for his doctorate under Tony Hoare as well at the PRG. Z is not explicitly named in the thesis, but an early document on Z by Jean-Raymond Abrial is referenced, as is a specification of a display editor using Z by Bernard Sufrin (actually Technical Monograph PRG-21). It is interesting to see the pioneers of formal methods who are referenced in the thesis, including the ACM A.M. Turing Award winners Edsger W. Dijkstra (1972), Tony Hoare (1980), and Amir Pnueli (1996).

Front page



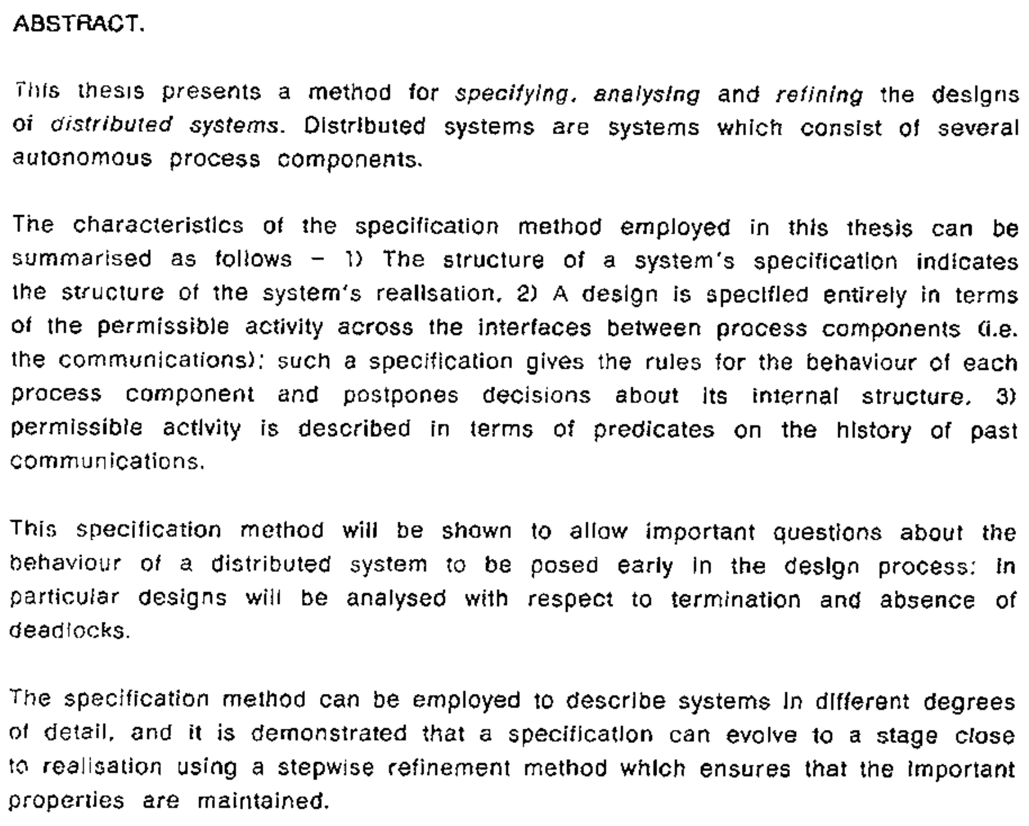
TOPICS IN PROGRAMME SPECIFICATION AND DESIGN:

SPECIFICATION AND DESIGN
OF
DISTRIBUTED SYSTEMS

I.H. SORENSEN
WOLFSON COLLEGE

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE UNIVERSITY OF OXFORD, SEPTEMBER 1981.

Abstract:



ABSTRACT.

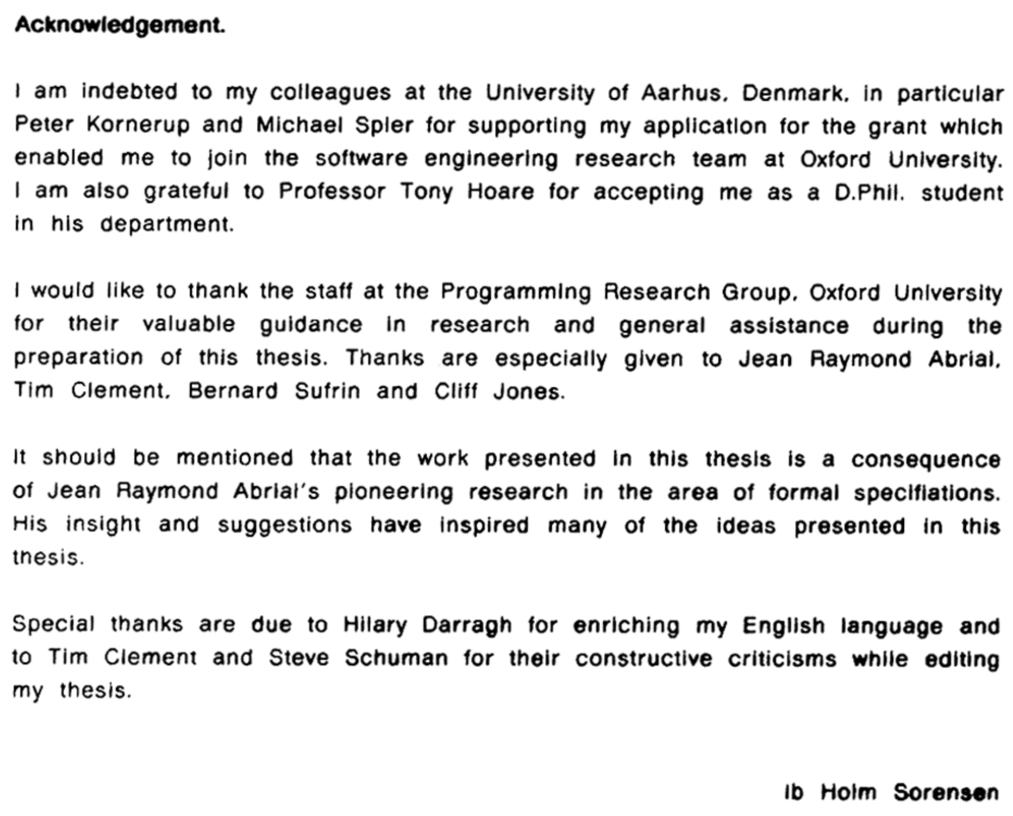
This thesis presents a method for *specifying, analysing* and *refining* the designs of *distributed systems*. Distributed systems are systems which consist of several autonomous process components.

The characteristics of the specification method employed in this thesis can be summarised as follows – 1) The structure of a system's specification indicates the structure of the system's realisation. 2) A design is specified entirely in terms of the permissible activity across the interfaces between process components (i.e. the communications); such a specification gives the rules for the behaviour of each process component and postpones decisions about its internal structure. 3) permissible activity is described in terms of predicates on the history of past communications.

This specification method will be shown to allow important questions about the behaviour of a distributed system to be posed early in the design process; in particular designs will be analysed with respect to termination and absence of deadlocks.

The specification method can be employed to describe systems in different degrees of detail, and it is demonstrated that a specification can evolve to a stage close to realisation using a stepwise refinement method which ensures that the important properties are maintained.

Acknowledgements:



Acknowledgement.

I am indebted to my colleagues at the University of Aarhus, Denmark, in particular Peter Kornerup and Michael Spier for supporting my application for the grant which enabled me to join the software engineering research team at Oxford University. I am also grateful to Professor Tony Hoare for accepting me as a D.Phil. student in his department.

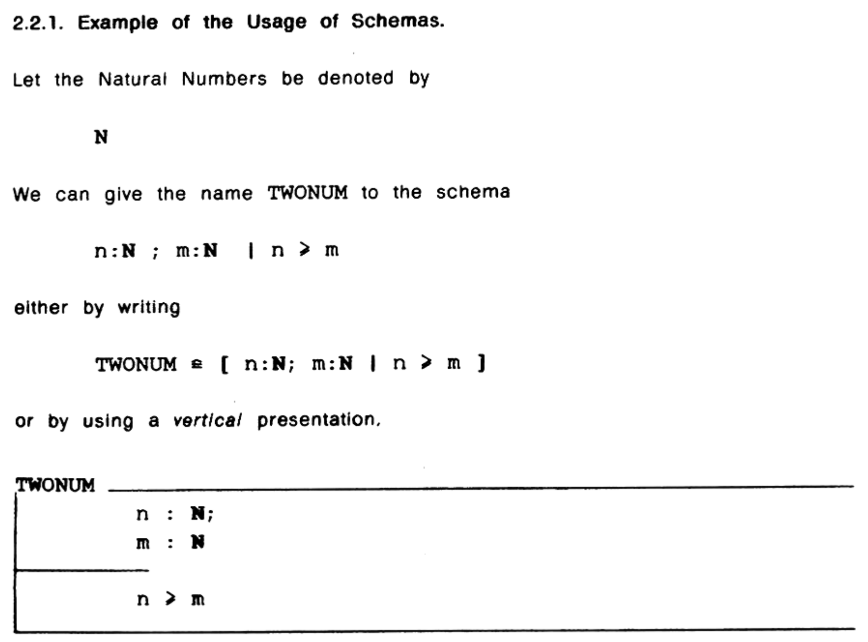
I would like to thank the staff at the Programming Research Group, Oxford University for their valuable guidance in research and general assistance during the preparation of this thesis. Thanks are especially given to Jean Raymond Abrial, Tim Clement, Bernard Sufrin and Cliff Jones.

It should be mentioned that the work presented in this thesis is a consequence of Jean Raymond Abrial's pioneering research in the area of formal specifiations. His insight and suggestions have inspired many of the ideas presented in this thesis.

Special thanks are due to Hilary Darragh for enriching my English language and to Tim Clement and Steve Schuman for their constructive criticisms while editing my thesis.

Ib Holm Sorensen

First example of a Z schema in the thesis:



2.2.1. Example of the Usage of Schemas.

Let the Natural Numbers be denoted by

$\mathbb{N}$

We can give the name TWONUM to the schema

$n:\mathbb{N} \; ; \; m:\mathbb{N} \;\; | \;\; n > m$

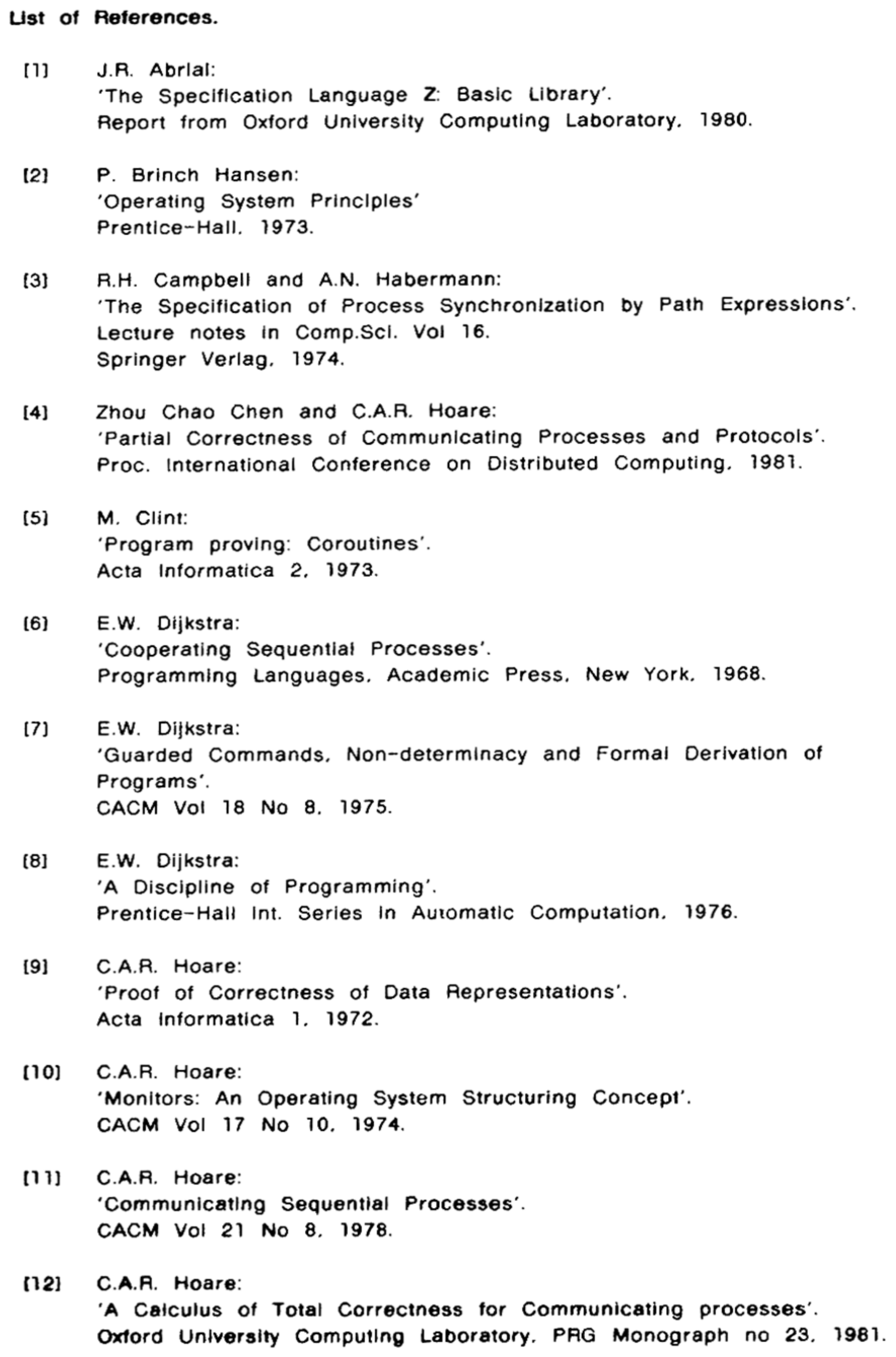
either by writing

$TWONUM \triangleq [\; n:\mathbb{N};\; m:\mathbb{N} \;|\; n > m \;]$

or by using a *vertical* presentation.

```
TWONUM ____________________________________
|       n : N;
|       m : N
|__________
|       n > m
|__________________________________________
```

List of references:



List of References.

[1] J.R. Abrial:
'The Specification Language Z: Basic Library'.
Report from Oxford University Computing Laboratory. 1980.

[2] P. Brinch Hansen:
'Operating System Principles'
Prentice-Hall. 1973.

[3] R.H. Campbell and A.N. Habermann:
'The Specification of Process Synchronization by Path Expressions'.
Lecture notes in Comp.Sci. Vol 16.
Springer Verlag. 1974.

[4] Zhou Chao Chen and C.A.R. Hoare:
'Partial Correctness of Communicating Processes and Protocols'.
Proc. International Conference on Distributed Computing. 1981.

[5] M. Clint:
'Program proving: Coroutines'.
Acta Informatica 2. 1973.

[6] E.W. Dijkstra:
'Cooperating Sequential Processes'.
Programming Languages. Academic Press. New York. 1968.

[7] E.W. Dijkstra:
'Guarded Commands. Non-determinacy and Formal Derivation of Programs'.
CACM Vol 18 No 8. 1975.

[8] E.W. Dijkstra:
'A Discipline of Programming'.
Prentice-Hall Int. Series in Automatic Computation. 1976.

[9] C.A.R. Hoare:
'Proof of Correctness of Data Representations'.
Acta Informatica 1. 1972.

[10] C.A.R. Hoare:
'Monitors: An Operating System Structuring Concept'.
CACM Vol 17 No 10. 1974.

[11] C.A.R. Hoare:
'Communicating Sequential Processes'.
CACM Vol 21 No 8. 1978.

[12] C.A.R. Hoare:
'A Calculus of Total Correctness for Communicating processes'.
Oxford University Computing Laboratory. PRG Monograph no 23. 1981.

[13] Reference Manual,
'Rationale for the Design of the GREEN Programming Language'
Honeywell, inc. Minneapolis, 1979.

[14] C.B. Jones:
'Software Developmant - A Rigorous Approach'.
Prentice-Hall Int. Series in Computer Science, 1980.

[15] C.B. Jones:
'Development method of Computer Programs Including a Notion of Interference'
D.Phil. thesis, University of Oxford, 1981.

[16] R.M. Keller:
'Formal Verification of Parallel Programs'.
CACM Vol 19 No 7, 1976.

[17] A. von Lamswerde and M. Sintzoff:
'Formal Derivation of Strongly Correct Concurrent Programs'.
Acta Informatica 12, 1979.

[18] Z.Manna and A. Pnuelli:
'Verification of Concurrent Programs: The Temporal Framework'.
Lecture notes for The International Summer School, Munich, August 1981.

[19] S. Owicki and D Gries:
'Verifying properties of Parallel Programs: An Axiomatic Approach'.
CACM Vol 19 No 5, 1976.

[20] P.L. Parnas:
'On the Criteria to be used in Decomposing Systems into Modules'.
CACM Vol 15 No 12, 1972.

[21] B. Sufrin:
'Formal Specification of a Display Editor'.
Oxford University Computing Laboratory, PRG Monograph no 21, 1981.

The Z User Group: Thirty Years After

Jonathan Bowen

July 2022

It is thirty years since the formation of the Z User Group (ZUG) [1], established to support the Z notation throughout the world [2].

The Z User Group was established in 1992 to promote the use and development of the Z notation, a formal specification language for the description of and reasoning about computer-based systems [3,4,5]. It was formally constituted on 14 December 1992 during the ZUM'92 Z User Meeting in London, England [6,7], at the instigation of John Nicholls.

The original Z User(s) Meeting (ZUM) was instigated by Ib Holm Sørensen at the Department of External Studies, Rewley House, University of Oxford, in 1985, under the auspices of the Programming Research Group, part of the Oxford University Computing Laboratory. However, there was no written report of the proceedings for this first meeting. Further meetings were held in the same location at Oxford in 1986 and 1987 with informally published proceedings [8,9]. The proceedings became formally published as the “Z User Workshop” in the Springer Workshops in Computing series for meetings in Oxford (1989 and 1990) [10,11], at the University of York (1991) [12], and at the Department of Trade and Industry in London (1992) [6], where the Z User Group was formally inaugurated.

After the establishment of the Z User Group, it continued to organise the Z User Meeting at St John’s College, Cambridge, in 1994 [13]. The Z User Meeting became the International Conference of Z Users in 1995, with the first conference held outside the UK, at the University of Limerick, Republic of Ireland, with the proceedings being published in the Springer Lecture Notes in Computer Science (LNCS) series [14]. Further conferences were held at the University of Reading, UK (1997) [15] and in Berlin, Germany (1998) [16]. The Z User Group participated at the FM'99 World Congress on Formal Methods in Toulouse, France, in 1999 [17].

In 2000, the Z conferences were merged to become the ZB Conference, jointly with the B-Method, co-organized with the Association de Pilotage des

Conférences B (APCB, aka the International B Conference Steering Committee), with the first conference in York, UK [18]. Subsequent ZB conferences were held in Grenoble, France (2002), Turku, Finland (2003), and Guildford, UK (2005). There were also additional Z User Meetings associated with the 2nd Systems and Software Week, Columbia, Maryland, USA, in April 2006, and the 12th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Auckland, New Zealand, in July 2007.

From 2008, the ZB Conference became the ABZ Conference, with Abstract State Machines as well at London South Bank University in London, UK [19]. In 2010, the ABZ Conference also included Alloy, a Z-like specification language with associated tool support in Orford, QC, Canada [20]. Subsequently, other state-based formal methods such as VDM (Vienna Development Method) from 2012 and TLA (Temporal Logic of Actions) from 2014 have been included as well. These further ABZ conferences have been held in Pisa, Italy (2012), Toulouse, France (2014), Linz, Austria (2016), Southampton, UK (2018), and Ulm Germany (2020 and 2021, combined online due to the pandemic, where the conference title was generalized to "Rigorous State-Based Methods" [21]). Information on papers in the proceedings for ZUM, ZB, and ABZ is available via DBLP online [22]. Covers of proceedings from 1987 to 2022 can be found in the appendix.

Successive chairs of the Z User Group have been as follows: John Nicholls (1992–1994); Jonathan Bowen (1994–2011); and Steve Reeves (from 2011). Successive secretaries have been the following: Mike Hinchey (1994–2011) and Randolph Johnson (from 2011). In 2011, the group and the associated Z notation were studied in the context of a Community of Practice [23]. Since then, in practice, the Z User Group has not been operational for the last decade, with ABZ conferences being supported by local institutions. ABZ 2023 is planned to be held in Nancy, France [2424].

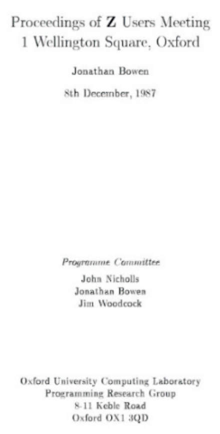
References

1. Sayeed, Ahmed (2021). Abbreviations. Sankalp Publication. p. 371. ISBN 978-9390636693.
2. Yearbook of International Organizations. Vol. 1. Union of International Associations. 2017.
3. Bowen, J.P. (September 1993). "Z User Group activities". JFIT News. 46: 5.
4. Bowen, J.P. (1994). "Z User Meeting Activities". High Integrity Systems. 1 (1): 93–94.
5. Tucker, Allen B., ed. (2004). Computer Science Handbook. CRC Press. Chapter 106. ISBN 978-0203494455.

6. Bowen, J.P.; Nicholls, J.E., eds. (1993). Z User Workshop, London 1992, Proceedings of the Seventh Annual Z User Meeting, 14–15 December 1992. Springer, Workshops in Computing, ISBN 978-3540198185.
7. "Z User Group (ZUG)". Global Civil Society Database. Union of International Associations (UIA).
8. Bowen, J.P., ed. (1987). Proceedings of Z Users Meeting. University of Oxford, UK, 8 December 1987. doi:10.13140/RG.2.2.20103.34724
9. Bowen, J.P., ed. (1988). Proceedings of the Third Annual Z Users Meeting. University of Oxford, UK, 16 December 1988.
10. Nicholls, J.E., ed. (1990). Z User Workshop, Oxford 1989, Proceedings of the Fourth Annual Z User Meeting, Oxford, UK, December 15, 1989. Springer, Workshops in Computing. ISBN 978-3540196273.
11. Nicholls, J.E., ed. (1991). Z User Workshop, Oxford 1990, Proceedings of the Fifth Annual Z User Meeting, 17–18 December 1990. Springer, Workshops in Computing. ISBN 978-3540196723.
12. Nicholls, J.E., ed. (1992). Z User Workshop, York 1991, Proceedings of the Sixth Annual Z User Meeting, 16–17 December 1991. Springer, Workshops in Computing. ISBN 978-3540197805.
13. Bowen, J.P.; Hall, J.A., eds. (1994). Z User Workshop, Cambridge 1994, Proceedings of the Eighth Annual Z User Meeting, 29–30 June 1994. Springer, Workshops in Computing. ISBN 978-3540198840.
14. Bowen, J.P.; Hinchey, M.G, eds. (1995). ZUM '95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7–9, 1995. Springer, Lecture Notes in Computer Science, Volume 967. ISBN 978-3540602712.
15. Bowen, J.P.; Hinchey, M.G.; Till, D., eds. (1997). ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3–4, 1997. Springer, Lecture Notes in Computer Science, Volume 1212. ISBN 978-3540627173.
16. Bowen, J.P.; Fett, A.; Hinchey, M.G., eds. (1998). ZUM '98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, September 24–26, 1998. Springer, Lecture Notes in Computer Science, Volume 1493. ISBN 978-3540650706.
17. "Z User Group Meeting (ZUG)". FM'99 World Congress. Toulouse, France. 20–24 September 1999.
18. Bowen, J.P.; Dunne, S.; Galloway, A.; King. S., eds. (2000). ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 – September 2, 2000.

Springer, Lecture Notes in Computer Science, Volume 1878. ISBN 978-3540679448.
19. Börger, E.; Butler, M.J.; Bowen, J.P.; Boca, P., eds. (2008). Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16–18, 2008. Springer, Lecture Notes in Computer Science, Volume 5238. ISBN 978-3540876021.
20. Frappier, M.; Glässer, U.; Khurshid, S.; Laleau, R.; Reeves, S., eds. (2010). Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22–25, 2010. Springer, Lecture Notes in Computer Science, Volume 5977. ISBN 978-3642118104.
21. Bowen, J.P. (2021). "ABZ 2021 Conference Report". FACS FACTS, 2021-2: 65–70, July.
22. "International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)". DBLP, Schloss Dagstuhl, Germany.
23. Bowen, J.P.; Reeves, S. (2011). "From a Community of Practice to a Body of Knowledge: A Case Study of the Formal Methods Community". In Butler, M.; Schulte, W. (eds.), FM 2011: Formal Methods. Springer, Lecture Notes in Computer Science, Volume 6664, pp. 308–322. doi:10.1007/978-3-642-21437-0_24
24. "ABZ 2023, 9th International Conference on Rigorous State Based Methods, May 30th to June 2nd 2023 – Loria, Nancy – France". Loria, France. https://abz2023.loria.fr

Appendix: Proceedings covers (1987 to 2021)

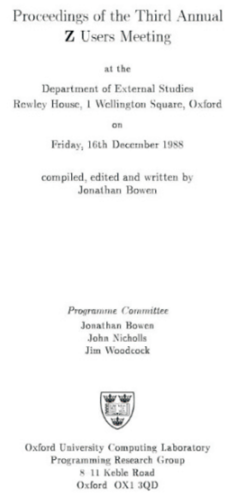

Proceedings of **Z** Users Meeting
1 Wellington Square, Oxford

Jonathan Bowen

8th December, 1987

Programme Committee
John Nicholls
Jonathan Bowen
Jim Woodcock

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD


Proceedings of the Third Annual
Z Users Meeting

at the

Department of External Studies
Rewley House, 1 Wellington Square, Oxford

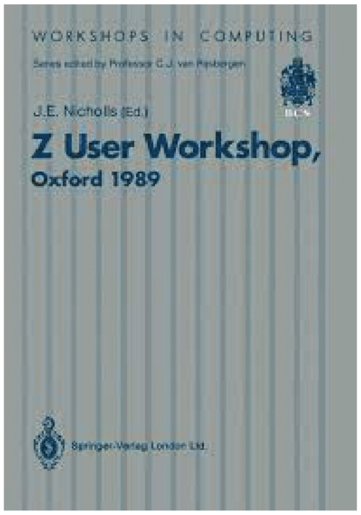
on

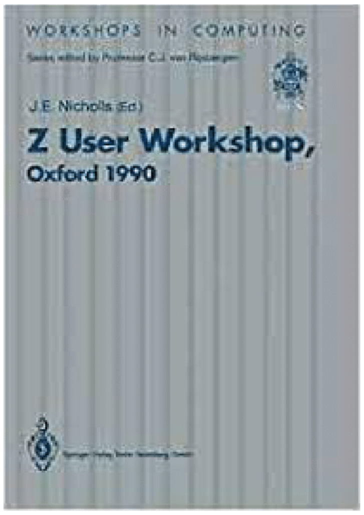
Friday, 16th December 1988

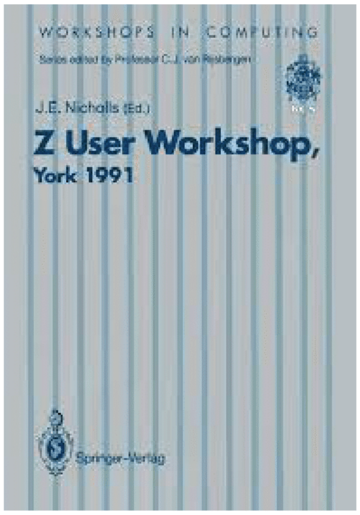
compiled, edited and written by
Jonathan Bowen

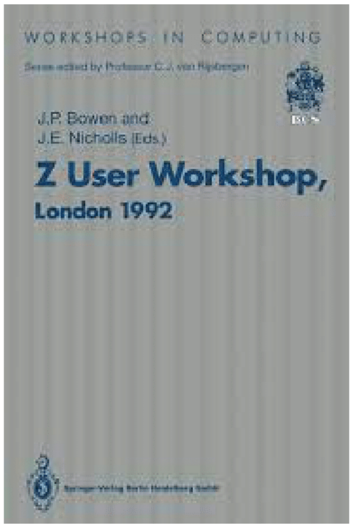
Programme Committee
Jonathan Bowen
John Nicholls
Jim Woodcock

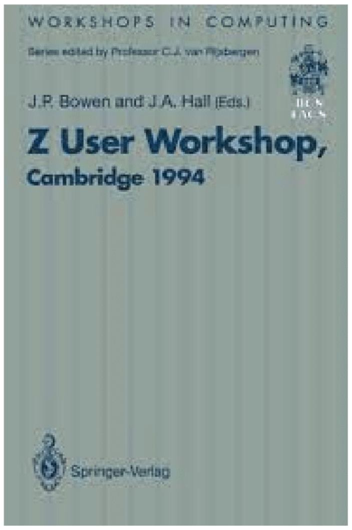
Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

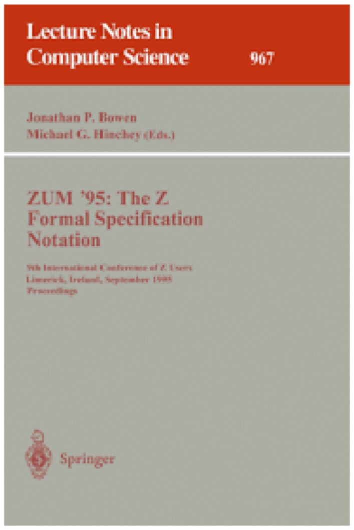

WORKSHOPS IN COMPUTING
Series edited by Professor C.J. van Rijsbergen
BCS
J.E. Nicholls (Ed.)
Z User Workshop,
Oxford 1989
Springer-Verlag London Ltd.


WORKSHOPS IN COMPUTING
Series edited by Professor C.J. van Rijsbergen
BCS
J.E. Nicholls (Ed.)
Z User Workshop,
Oxford 1990
Springer-Verlag Berlin Heidelberg GmbH


WORKSHOPS IN COMPUTING
Series edited by Professor C.J. van Rijsbergen
BCS
J.E. Nicholls (Ed.)
Z User Workshop,
York 1991
Springer-Verlag


WORKSHOPS IN COMPUTING
Series edited by Professor C.J. van Rijsbergen
BCS
J.P. Bowen and
J.E. Nicholls (Eds.)
Z User Workshop,
London 1992
Springer-Verlag Berlin Heidelberg GmbH


WORKSHOPS IN COMPUTING
Series edited by Professor C.J. van Rijsbergen
BCS FACS
J.P. Bowen and J.A. Hall (Eds.)
Z User Workshop,
Cambridge 1994
Springer-Verlag

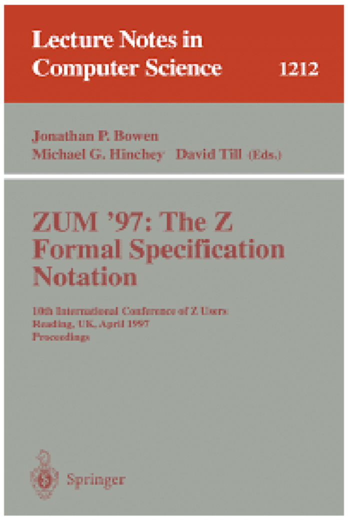

Lecture Notes in
Computer Science 967

Jonathan P. Bowen
Michael G. Hinchey (Eds.)

ZUM '95: The Z
Formal Specification
Notation

9th International Conference of Z Users
Limerick, Ireland, September 1995
Proceedings

Springer

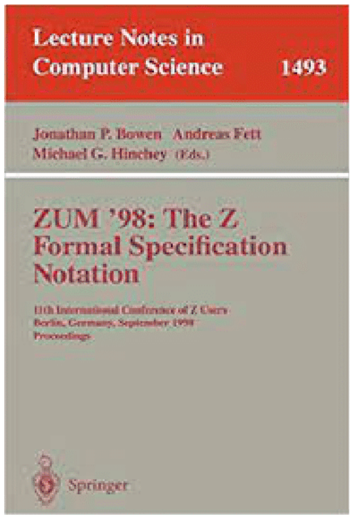

Lecture Notes in Computer Science 1212

Jonathan P. Bowen Michael G. Hinchey David Till (Eds.)

ZUM '97: The Z Formal Specification Notation

10th International Conference of Z Users
Reading, UK, April 1997
Proceedings

Springer

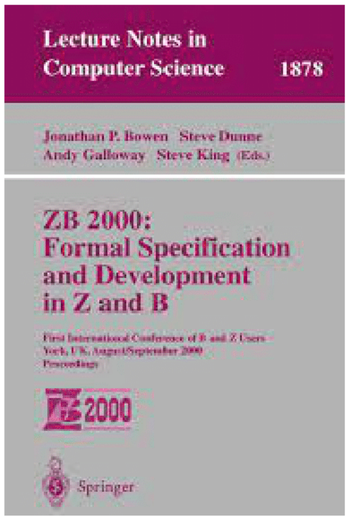

Lecture Notes in Computer Science 1493

Jonathan P. Bowen Andreas Fett Michael G. Hinchey (Eds.)

ZUM '98: The Z Formal Specification Notation

11th International Conference of Z Users
Berlin, Germany, September 1998
Proceedings

Springer


Lecture Notes in Computer Science 1878

Jonathan P. Bowen Steve Dunne Andy Galloway Steve King (Eds.)

ZB 2000: Formal Specification and Development in Z and B

First International Conference of B and Z Users
York, UK, August/September 2000
Proceedings

ZB 2000

Springer

Didier Bert
Jonathan P. Bowen
Martin C. Henson
Ken Robinson (Eds.)

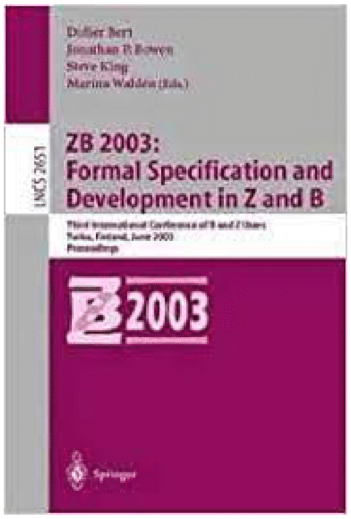
LNCS 2272

ZB 2002: Formal Specification and Development in Z and B

2nd International Conference of B and Z Users
Grenoble, France, January 2002, France
Proceedings

ZB 2002

Springer


Didier Bert
Jonathan P. Bowen
Steve King
Marina Waldén (Eds.)

LNCS 2651

ZB 2003: Formal Specification and Development in Z and B

Third International Conference of B and Z Users
Turku, Finland, June 2003
Proceedings

ZB 2003

Springer

Helen Treharne
Steve King
Martin Henson
Steve Schneider (Eds.)

LNCS 3455

ZB 2005: Formal Specification and Development in Z and B

4th International Conference of B and Z Users
Guildford, UK, April 2005
Proceedings

Springer

Egon Börger
Michael Butler
Jonathan P. Bowen
Paul Boca (Eds.)

LNCS 5238

Abstract State Machines, B and Z

First International Conference, ABZ 2008
London, UK, September 2008
Proceedings

Springer

Marc Frappier Uwe Glässer
Sarfraz Khurshid Régine Laleau
Steve Reeves (Eds.)

LNCS 5977

Abstract State Machines, Alloy, B and Z

Second International Conference, ABZ 2010
Orford, QC, Canada, February 2010
Proceedings

Springer

John Derrick John Fitzgerald
Stefania Gnesi Sarfraz Khurshid
Michael Leuschel Steve Reeves
Elvinia Riccobene (Eds.)

LNCS 7316

Abstract State Machines, Alloy, B, VDM, and Z

Third International Conference, ABZ 2012
Pisa, Italy, June 2012
Proceedings

Springer

Yamine Ait Ameur
Klaus-Dieter Schewe (Eds.)

LNCS 8477

Abstract State Machines Alloy, B, TLA, VDM, and Z

4th International Conference, ABZ 2014
Toulouse, France, June 2–6, 2014
Proceedings

Springer

Michael Butler · Klaus-Dieter Schewe
Atif Mashkoor · Miklos Biro (Eds.)

LNCS 9675

Abstract State Machines, Alloy, B, TLA, VDM, and Z

5th International Conference, ABZ 2016
Linz, Austria, May 23–27, 2016
Proceedings

Springer

Michael Butler · Alexander Raschke
Thai Son Hoang · Klaus Reichl (Eds.)

LNCS 10817

Abstract State Machines, Alloy, B, TLA, VDM, and Z

6th International Conference, ABZ 2018
Southampton, UK, June 5–8, 2018
Proceedings

Springer

Alexander Raschke
Dominique Méry
Frank Houdek (Eds.)

LNCS 12071

Rigorous State-Based Methods

7th International Conference, ABZ 2020
Ulm, Germany, May 27–29, 2020
Proceedings

Springer

Alexander Raschke
Dominique Méry (Eds.)

LNCS 12709

Rigorous State-Based Methods

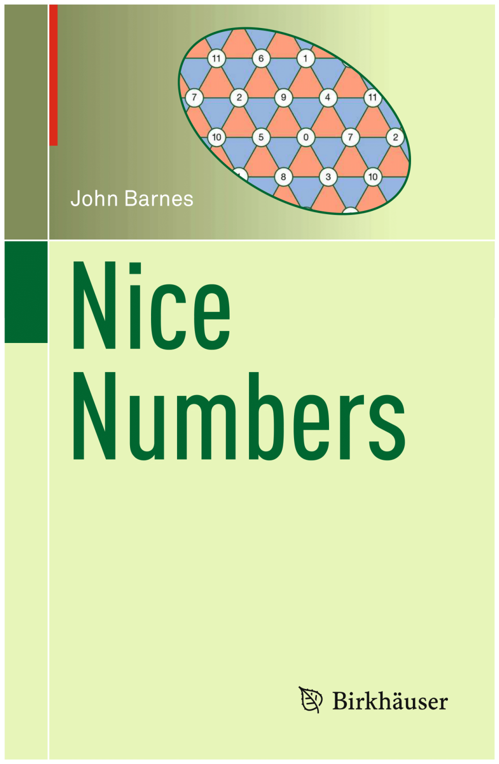
8th International Conference, ABZ 2021
Ulm, Germany, June 9–11, 2021
Proceedings

Springer

Review of *Nice Numbers* by John Barnes

Springer International (Birkhäuser) 2016

Reviewed by: Tim Denvir
July 2022



First of all, note that this is the John Barnes who has done much work on Ada and other programming languages, ***not*** the US American science fiction writer of the same name!

Nice Numbers is based on notes for a series of lectures which the author gave to adult education classes in Oxford and Reading. It is therefore not a text that might be used for an academic qualification, but more aimed at a mathematical hobbyist. Everything is defined and explained. Nonetheless I think a reader would need at least the equivalent of O-level (or GCSE) and will have to be in tune with mathematical thinking a fair way beyond that, perhaps to A-level standard, to follow John Barnes's exposition, even though no knowledge is assumed. But I absolutely don't want to put anyone off: this is an intensely pleasurable read, energised by the author's infectious exuberance and wit.

Especially given the above, it is a pity that the book is priced so high - £49.99 RRP, £39.99 downloadable pdf version. The least expensive hardcopy I could find was £31.79 from WOB (World of Books). The RRP from the publisher curiously is the same for hard and softback. I feel sure they would sell dramatically more copies, increasing the profit for themselves (and royalties for the author!) if they reduced the price to £26 or so. The publisher would only supply me with an electronic copy for review, departing from normal practice. Since then I have obtained a hardback copy by another route, and I can say that the physical quality of the book is excellent: printed on high quality glossy paper and with very clear fine-edged print. I haven't seen a softback copy, but who would want to buy one when the hard and softbacks are the same price?

You may wonder why a book like this might be considered of interest to the FACS community. In the chapters where the author discusses bases, the base of 2 has special properties and of course is highly relevant to digital computing. However, my main excuse is that everyone who delves into formal aspects of computing comes to grips with quite a lot of mathematics, much of which is outside the traditional curricula, and so usually has to be gathered from scratch. Despite being about numbers, which we all think we know about, this book contains a lot which will be new and fascinating even to those well versed in mathematics. The book does not just explore today's understanding of numbers, but makes comparison with ancient Egyptian and Babylonian methods of calculation and numeric notation: an erudite account of knowledge evolution through history. The author also goes into the etymology of quite a lot of familiar mathematical terms, which highlights the thinking behind their origins, something I especially enjoyed.

Chapter 1 Measures

John Barnes starts off his courses by asking people what are their favourite numbers. This chapter lists the various answers with the reasons examined. 7 was the most popular. He then goes on to talk about primes, shows a way of finding new primes, Euclidean primes, then various unsolved conjectures about primes, such as whether primes 2 apart go on for ever. Then there is a section on factors, and deficient, abundant, superabundant, perfect numbers. Some interesting facts, such as the first few factorials are superabundant, but 8! is not (a superabundant number is more abundant than all its predecessors).

Weights and Measures

This section takes me back. As a nine year-old in a rural village school we all had to learn the archaic imperial measures: 4 poles in a chain, which was also 100 links, 10 chains in a furlong, a furlong being one eighth of a mile, in other words 220 yards which meant that a chain was 22 yards, the length of a cricket pitch. I remember seeing an actual physical chain consisting of 100 links, used to mark out where the opposing stumps were placed. And then there were bushels, pecks, quarts, only a bushel was a different size depending on whether it was a bushel of wheat or something else. But John Barnes' knowledge of archaic weights and measures puts mine to shame. He relates the many different scales to potential bases; it is a tragedy that we have ten fingers and toes, he says, because 10 is not a good base, having few factors. 12 would be better, and explains why it features in many scales (inches to a foot, and a dozen, a gross etc.). He points out that the troy ounce is more than the avoirdupois ounce, but the troy pound is less than the avoirdupois pound! I had not twigged before that a square chain is one tenth of an acre.

His excursion into currencies, both British and continental is fascinating.

The well known rules of thumb for quickly finding if a number is divisible by certain factors, 3, 9, 11 and 4 in base 10 are not mentioned here as I was expecting, but in a later chapter. Similar corresponding rules work for other bases.

Chapter 2 Amicable Numbers

More about perfect numbers, Mersenne numbers. John peppers his text with a characteristic wit, which makes the reading more pleasurable: e.g. “What is the point in finding large primes? Until recently it was just for fun like climbing Mt Everest.” Whereas a perfect number is the sum of its factors, Amicable Numbers are pairs of numbers each of which is the sum of the factors of the other. Sociable numbers are chains of numbers with this property. Modular arithmetic is important and useful for proving properties of these kinds of numbers. He defines Fermat numbers and Fibonacci numbers.

Chapter 3 Probability

The author goes into careful detail about the distinction between probability and statistics. He explains gambling games, poker, craps and “double or quits”, and reveals intricate details about the design of dice beyond that numbers on opposite sides should add up to 7. Slightly involved in places but very good.

Chapter 4 Fractions

He goes into intriguing (and possibly too great) detail about Egyptian multiplication and division. Certainly it has historical value. We are reminded of the tedious manual methods of finding square roots and cube roots. There is a lot about ancient Egyptian methods. Decimal fractions and Continued fractions, resurrected from the early 19th century. I felt glad I wasn’t a schoolboy in ancient Egyptian times!

Chapter 5 Time

John Barnes gives an astronomical description of days, months, years. Despite my having been an enthusiastic student of astronomy from the age of 6, there were some details about the variations in length of the sidereal day that I did not know about. There is much history of the Julian and Gregorian calendars, quarter days etc., most illuminating! The involved calculations about the different amounts of energy from the sun on the earth’s surface show remarkable and surprising results.

Chapter 6 Notation

He compares Roman, Arabic, Egyptian and Babylonian numeral systems, and categorises them. He discusses place systems and bases, particularly considering fractions in different bases. He shows recurring cycles of digits in different bases and derives rules about them. He finally returns to Fermat’s Little Theorem.

Chapter 7 Bells

The author relates bell ringing to permutations. Again, as in all the previous chapters, he provides historical details: for example, Fabian Stedman (1640-1713) wrote two famous books on bell ringing, *Tintinnalogia*, and *Campanalogia*. This chapter goes into intricate detail and would require dedication to read thoroughly and grasp its entirety. It ends with a couple of paragraphs indicating that the permutations of bell sequences form an algebraic group, and gives references to more in-depth analyses, including Appendix E: Groups.

Chapter 8 Primes

This starts out with Greatest Common Divisors (GCD) also known as Highest Common Factors (HCF) and relates them to Fibonacci Numbers and gives fast methods for finding factors of large numbers (Eratosthenes and Fermat). Then he embarks on complex numbers, expansions of e^x and sin x and cos x. Complex primes are defined. The chapter ends with a short section on polynomials including prime polynomials.

Chapter 9 Music

This presents the basic physics of vibrating strings and columns of air in pipes, musical intervals, chromatic and diatonic semitones and how they arise. Different kinds of scales, how they all deviate from the ideal, which is impossible to achieve. Scales where C# is different from D♭, major and minor scales, frequencies, and all their histories. All are mathematically analysed. Anyone with an interest in music who has a mathematical bent will find this chapter interesting.

Chapter 10 Finale

This final chapter is a miscellany of topics: the use of primes in encryption, the RSA algorithm, which includes linear congruences and Diophantine equations; animal gaits, bipedal and quadrupedal; the games of Towers of Hanoi and, related, Chinese Rings.

Appendices

Finally, there are nine appendices covering various topics including Ackermann's function, Stochastics, Groups, and Rubik's cube (!).

Forthcoming events

Events Venue (unless otherwise specified):

> BCS, The Chartered Institute for IT
> Ground Floor, 25 Copthall Avenue, London, EC2R 7BP

The nearest tube station is Moorgate, but Bank and Liverpool Street are within walking distance as well. The new Elizabeth Line is now very convenient for the BCS London office, by alighting at the Liverpool Street stop and leaving via the Moorgate exit.

Details of all forthcoming events can be found online here:

> https://www.bcs.org/membership/member-communities/facs-formal-aspects-of-computing-science-group/

Please revisit this site for updates as and when further events are confirmed.

FACS Committee

Formal Aspects of Computing Science Specialist Group

Jonathan Bowen
FACS Chair and BCS Liaison

John Cooke
FACS Treasurer and Publications

Roger Carsley
Minutes Secretary

Andrei Popescu
LMS Liaison

Ana Cavalcanti
FME Liaison

Brijesh Dongol
Refinement Workshop Liaison

Keith Lines
Government and Standards Liaison

Margaret West
Inclusion Officer and BCS Women Liaison

Tim Denvir
Co-Editor, FACS FACTS

Brian Monahan
Co-Editor, FACS FACTS

FACS is always interested to hear from its members and keen to recruit additional helpers. Presently we have vacancies for officers to help with fund raising, to liaise with other specialist groups such as the Requirements Engineering group and the European Association for Theoretical Computer Science (EATCS), and to maintain the FACS website. If you are able to help, please contact the FACS Chair, Professor Jonathan Bowen at the contact points below:

BCS-FACS
c/o Professor Jonathan Bowen (Chair)
London South Bank University
Email: jonathan.bowen@lsbu.ac.uk
Web: www.bcs-facs.org

You can also contact the other Committee members via this email address.

Mailing Lists

As well as the official BCS-FACS Specialist Group mailing list run by the BCS for FACS members, there are also two wider mailing lists on the Formal Aspects of Computer Science run by JISCmail.

The main list <facs@jiscmail.ac.uk> can be used for relevant messages by any subscribers. An archive of messages is accessible under:

http://www.jiscmail.ac.uk/lists/facs.html

including facilities for subscribing and unsubscribing.

The additional <facs-event@jiscmail.ac.uk> list is specifically for announcement of relevant events.

Similarly, an archive of announcements is accessible under:

http://www.jiscmail.ac.uk/lists/facs-events.html

including facilities for subscribing and unsubscribing.

BCS-FACS announcements are normally sent to these lists as appropriate, as well as the official BCS-FACS mailing list, to which BCS members can subscribe by officially joining FACS after logging onto the BCS website.