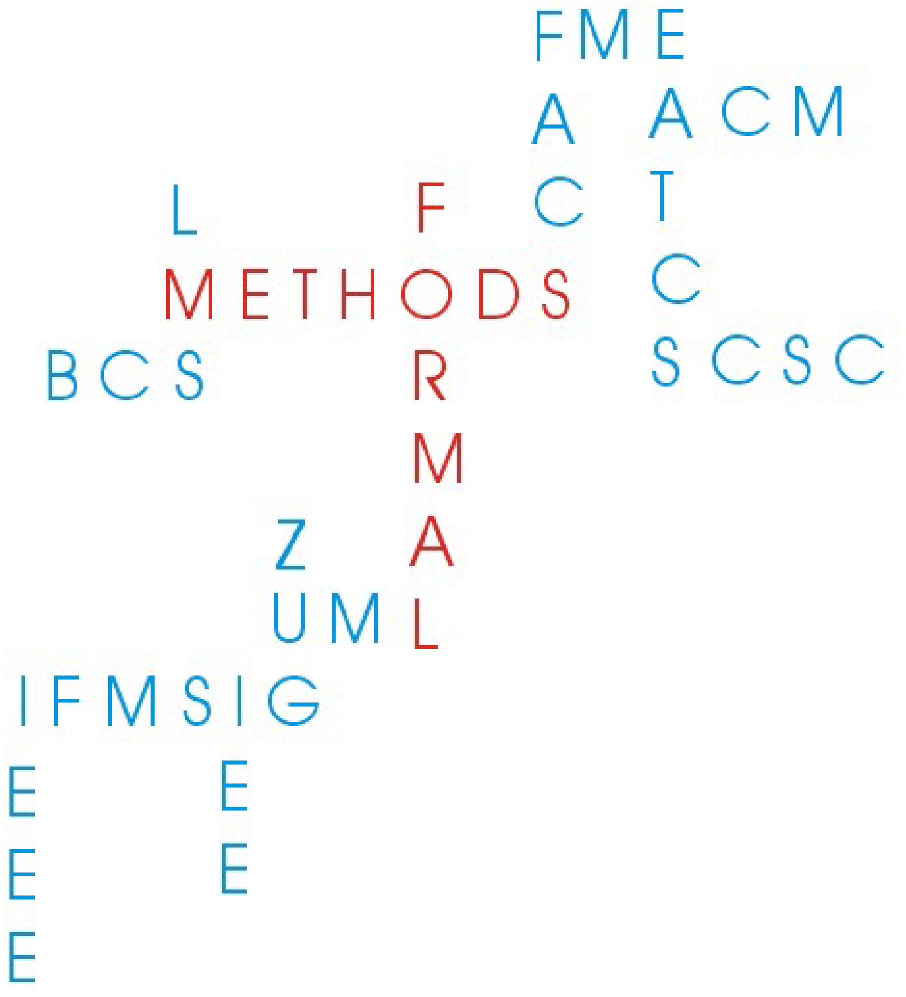
Issue 2006-2
September 2006

FACS FACTS



BCS FACS

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.

FACS FACTS is published up to four times a year: **March**, **June**, **September** and **December**. Submissions are always welcome. Please see the advert on **page 12** for further details or visit the newsletter area of the FACS website [http://www.bcs-facs.org/newsletter].

Back issues of *FACS FACTS* are available to download from:

http://www.bcs-facs.org/newsletter/facsfactsarchive.html

The *FACS FACTS* Team

Newsletter Editor	Paul Boca [editor@facsfacts.info]
Editorial Team	Jonathan Bowen, Judith Carlton, John Cooke, Kevin Lano
Columnists	Dines Bjørner (Train Domain)

Contributors to this Issue

Joan Atkinson, Paul Boca, Jonathan Bowen, Steve Dunne, John Fitzgerald, Cliff Jones, Ian Hayes, Tiziana Margaria, Ivana Mijajlovic, Faron Moller, Ken Pierce, John Tucker, F.X. Reid, Bernard Schätz, Marcel Verhoef, Phan Cong Vinh

Contents

The activities of FACS (i.e. **sponsoring conferences and workshops**, **offering student bursaries** and **hosting evening seminars**) is funded solely from membership subscriptions. The more paid-up FACS members we have, the more we can do. ☺

If you would like to become a FACS member – or renew your lapsed membership – please complete the membership form on **page 83** of this issue of *FACS FACTS*.

If you have any questions about FACS, please send these to Paul Boca [Paul.Boca@virgin.net], the Membership Secretary.

Editorial

Paul Boca & Jonathan Bowen, BCS-FACS

We must start off this issue of the newsletter with an apology: in Issue 2006-1 we incorrectly reported the death of long-time contributor to *FACS FACTS*, F.X. Reid. The obituary, while interesting in shedding light on FXR's life, was utter nonsense. Victor Zemantics' motives for submitting the obituary remain a mystery, as do the exact whereabouts of FXR. FXR has made contact with the editor of *FACS FACTS*, and his letter (unedited) appears below:

Dear Professor Boca,

I am contacting you to inform you and the rest of my fans that I did not, as reported, go over the Lautenbach Falls, or whatever was reported in your ridiculous 'obituary'. Nor would I ever contemplate living in Marsascala - don't you people ever read the Maltese times? I am alive and well and living in San Pawl Il Bahar. To find my address please apply to my advocate.

The truth is that I retired from academia from disgust of your pusilanimousness. 23% What! You're worth twice that! Get on to the barricades and show the bureaucrats who's boss! 60% or nothing! And besides, just think how it will annoy the students! (And don't go po-faced on me.)

I am busy completing my definitive treatise on why all the interleavers are wrong, and will be delivering the main points of my analysis later this year – probably in the Albert Hall. Formal dress only please.

But it has been at least an hour before I sat beside the shore, looking out over St Paul's Islands and drinking a planters punch, so I sign off now. Enjoy the weather.

F. X. Reid

FXR has accepted our sincere apology (and a lifetime membership to FACS!) for the error and I hope he will continue to send in submissions, assuming that is we can lure him away from the punch! Incidentally, FXR, your addressee is not a Professor, and never will be, but thanks for the meteoric rise; he's just a POD – Plain Ol' Doctor of the non inter-galactic variety! In any case, we will be much more careful with our sources (sauces?!) in future. FXR's Wikipedia entry has of course been suitably updated [http://en.wikipedia.org/wiki/F._X._Reid].

Back to business: the FACS evening seminars continue to go from strength to strength. The quality of the seminars has been outstanding, and attendance figures greater than we had imagined. FACS would like to thank the speakers and attendees for their support. The organizers, Paul Boca, Jonathan Bowen and Jawed Siddiqi, are currently putting together a book based on the seminars to be published by Springer in 2007.

FACS is currently taking a short break from the seminars over the summer period, but will return on **4 September** with *The Computer Ate my Vote*, which will be delivered by Professor Peter Ryan, University of Newcastle. Professor Ursula Martin, Queen Mary University of London, will deliver the final seminar of the year on **9 November**. We have already started putting together seminars for 2007. Two dates already fixed are:

- Michael Jackson, independent consultant, **7 February 2007**
- Professor Egon Börger, University of Pisa, **21 March 2007**

This edition of the newsletter contains a report on *Formal Methods in the Last 25 Years*, which you will recall was the first evening seminar of 2006, held jointly with Formal Methods Europe. The event was audio recorded, and a summary of the transcript is included in that report. A reprint of the *Magic Roundabout* paper appears in this issue too, as well as an extract from a paper by Tim Clement and Cliff Jones on *Model-Oriented Specifications*.

Reports on the British Colloquium for Theoretical Computer Science (BCTCS) and workshop on Unified Theories of Programming appear too. FACS sponsored both events. The BCTCS report has details of a new learned society for theoretical computer science, which we hope some readers will find of interest. FACS will keep you posted on any further developments in this area.

A report on the industry day that took place at Formal Methods 05 appears in this issue too. It contains a good summary of real-life uses of formal methods – definitely worth reading.

We hope you will enjoy this edition of the newsletter and will consider contributing articles to future issues. Without contributions, there is no newsletter! We hope to see you at one of the upcoming seminars too.

The Electronic Workshops in Computing series (eWiC) is published by the British Computer Society and includes the proceedings of workshops and conferences on the broadest possible range of computing topics. eWiC gives ***FREE*** access to over 40 workshop proceedings.

NEW PROCEEDINGS ON eWiC:

Workshop on Mathematically Structured Functional Programming, MSFP '06, Kuressaare, Estonia, 2 July 2006.

Programme Co-chairs

Conor McBride (University of Nottingham, UK)
Tarmo Uustalu (Institute of Cybernetics, Estonia)

http://www.bcs.org/server.php?show=nav.00100v005003001001002

UTP01 – 1st International Symposium on Unified Theories of Programming

Steve Dunne

Background

This first international symposium on Unifying Theories of Programming (UTP) took place on 5-7 February 2006 at the Walworth Castle just outside Darlington in County Durham. The aim of the symposium was to bring together academics actively engaged in research in the area of formal notations and theories used in the description of abstract systems and formal software development. The symposium encouraged contributions pursuing the UTP agenda, as initiated by Hoare & He in their 1998 book *Unifying Theories of Programming*, which seeks to put all the various theories which have emerged and become widely used into a common semantic basis to facilitate their comparison and combined use.

UTP was attended by some 32 delegates from as far afield as Australia, Brazil and Canada as well as France, Germany and Ireland, and from leading UK universities such as Durham, Oxford, Surrey and York. The invited speakers were Professor Ian Hayes from Queensland, Professor Eric Hehner from Toronto, Sir Tony Hoare from Microsoft Research, Dr Jeff Sanders from Oxford University and Professor Jim Woodcock from the University of York.

Submissions & Publications

The symposium received 17 submissions in response to the Call for Papers, a very satisfactory figure given the specialized nature of the symposium, the newness of the event and the relatively short submission lead-time. Nine of the submitted papers were selected for presentation at the symposium and inclusion in the proceedings, in addition to the contributions of the five invited speakers. A Springer LNCS volume of post-proceedings papers appeared in May 2006 (Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming*, Springer LNCS 4010, 2006). In addition, an associated special issue of the journal *Formal Aspects of Computing* featuring extended versions of some of these papers will appear in 2007.

Sunday 5 February

After registration and welcome, the symposium programme commenced at 1630 hrs with a short preliminary session of talks outside the main proceedings. The first of these was given by Andrew Butterfield of Trinity College Dublin, and the second by Bernhard Moeller of the University of Augsburg. Both talks triggered interesting discussions. The slides of Dr Butterfield's talk are available on the symposium website, while Professor Moeller's talk was accompanied by a draft 20-page paper which was distributed to all delegates. In the evening most delegates convened in the Castle's restaurant to continue their discussions informally over dinner.

Monday 6 February

The first full session of the symposium commenced with Eric Hehner's invited talk which provided a keynote for the whole symposium. This was followed by a presentation by Thiago Santos of the University of Pernambuco. Next came Sir Tony Hoare's invited talk, and then a presentation by Bill Stoddart of Teesside. The afternoon session opened with Jim Woodcock's invited talk followed by presentations by Shengchao Qin of Durham and Marcel Oliveira of York.

In the evening delegates convened again in the Castle's restaurant for the *de facto* symposium dinner, at the end of which the Symposium Chair made a short impromptu speech.

Tuesday 7 February

The final day of the symposium commenced with Ian Hayes' invited talk, followed by a presentation by Yifeng Chen of Durham. This in turn was followed by Jeff Sanders' invited talk and then by presentations from Will Harwood of Citrix UK, Gift Nuka of Kent and Walter Guttmann of Ulm.

A general discussion among delegates took place at the end of the symposium, from which it was clear there was a strong wish that the event be repeated periodically in the future. Accordingly it is anticipated that another UTP symposium will be organized to take place in two years time.

Conclusion

This symposium was clearly appreciated and enjoyed by all its participants. Several commented that what they really appreciated was that all the talks were interesting and relevant. This was doubtless owing to the coherence the symposium enjoyed by being entirely focused on the strong single theme of UTP. One delegate said “There was an unusually high bandwidth of communication between the participants." The event format of two complete days in a residential venue worked well in generating a positive and friendly atmosphere in which many productive conversations took place outside the scheduled sessions. It was clear from the plenary discussion at the end that the UTP community shares an appetite for further events like this one.

If you would like to become a FACS member, please complete the form on page 83 and return it to the FACS Membership Secretary with your membership fee.

By joining now, you will qualify for a **reduced entry fee** at the upcoming FACS Christmas meeting on Teaching Formal Methods (see page 78 for details)

The Safety-Critical Systems Club

http://www.safety-club.org.uk

What is the Club?

The Safety-Critical Systems Club exists to raise awareness and facilitate technology transfer in the field of safety-critical systems. It is a non-profit organization that cooperates with all bodies involved with or interested in safety-critical systems.

History

The Club was inaugurated in 1991 under the sponsorship of the Department of Trade and Industry (DTI) and the Engineering and Physical Sciences Research Council (EPSRC), and is organised by the Centre for Software Reliability (CSR) at the University of Newcastle upon Tyne. Its Co-ordinator is Felix Redmill of Redmill Consultancy.

Since 1994 the Club has had to be self-sufficient, but it retains the active support of the DTI and EPSRC, as well as that of the Health and Safety Executive, the Institution of Electrical Engineers, and the British Computer Society. All of these bodies are represented on the Club's Steering Group.

What does the Club do?

The Club achieves its goals of technology transfer and awareness raising by focusing on current and emerging practices in safety engineering, software engineering, and standards which relate to safety in processes and products. Its activities include:

- Running the annual Safety-critical Systems Symposium each February (of which there have been eight), with Proceedings published by Springer-Verlag;
- Organising a number of 1- or 2-day seminars each year;
- Providing tutorials on relevant subjects;
- Publishing a newsletter, *Safety Systems*, three times each year (since 1991) — in January, May and September.

How does the Club help?

The Club brings together technical and managerial personnel within all sectors of the safety-critical community. It facilitates communication among researchers, the transfer of technology from researchers to users, feedback from users, and the communication of experience between users. It provides a meeting point for industry and academia, a forum for the presentation of the results of relevant projects, and a means of learning and keeping up-to-date in the field.

The Club thus helps to achieve more effective research, a more rapid and effective transfer and use of technology, the identification of best practice, the definition of requirements for education and training, and the dissemination of information.

Membership

Members pay a reduced fee (well below a commercial rate) for events and receive the newsletter and other mailed information. Without sponsorship, the Club depends on members' subscriptions, which can be paid at the first meeting attended. As a recognised industrial community Club the charges levied are paid – in almost all cases – by employers; corporate arrangements are available on request.

To join, please contact Joan Atkinson at:

CSR, Claremont Tower, University of Newcastle upon Tyne, NE1 7RU
Telephone: +44 191 221 2222;
Fax: +44 191 222 7995;
Email: csr@newcastle.ac.uk
http://www.csr.ncl.ac.uk

The Ideal of Program Correctness

The Computer Journal presents: The Ideal of Program Correctness
by Tony Hoare.

Wednesday 25th October 2006

5:30pm for a 6:00pm start

BCS London Offices

Lecture by Tony Hoare - Microsoft Research, Cambridge. The lecture will be followed by a debate. The lecture itself, and the discussion following it, will be edited for the Computer Journal.

Tony Hoare will propose that program correctness is a basic scientific ideal for Computer Science, and should be pursued for its own sake.

In that way, one can accumulate a body of relevant scientific theory, supported by widely ranging experiment, and implemented in an integrated programming toolset to be used by the professional programmer.

For further information, please visit the BCS website:

http://www.bcs.org/server.php?show=ConWebDoc.6746

(Note: This is not a BCS-FACS Event)

CSR

2006 CALENDAR OF EVENTS

Provisional

SAFETY-CRITICAL SYSTEMS CLUB MEETINGS

Safety-critical Systems Symposium '06	7–9 February Bristol
Ada Conference UK 2006	28 March Manchester
Safety Cases – from Cradle to Grave	9 May London
The State of the Practice in Safety Systems	6 July London
Principles of Safety Engineering & Management	17 October London
Tutorial by Felix Redmill, *Redmill Consultancy* Safety Analysis, Tutorial by Prof. Peter Ladkin, *Causalis Ltd*	18 October
Value Added by Independent Safety Assessment Jointly with ISA Working Group	7 December London
Safety-critical Systems Symposium '07	13–15 February Bristol

Details of our 2006 events can be found at:

www.safety-club.org.uk

BCTCS 2006 — 22nd British Colloquium for Theoretical Computer Science

Faron Moller

The British Colloquium for Theoretical Computer Science (BCTCS) is an annual forum for researchers in theoretical computer science to meet, present research findings, and discuss developments in the field. It also provides an environment for PhD students to gain experience in presenting their work in a wider context, and benefit from contact with established researchers.

BCTCS 2006 was held at Swansea University during 4–7 April 2006. The event attracted 122 participants, and featured an interesting and wide-ranging programme of 6 invited talks and 62 contributed talks; roughly half of the participants and speakers were PhD students. Abstracts and slides for the talks are available from the BCTCS website [http://www.bctcs.ac.uk]. The financial support of the Engineering and Physical Sciences Research Council (EPSRC), the Welsh Development Agency (WDA) the London Mathematical Society (LMS), the British Computer Society (BCS), and the BCS Formal Aspects of Computing Science Specialist Group (FACS) is gratefully acknowledged.

A novel feature of BCTCS 2006 was the form of its opening Invited Lecture, which came in the guise of Peter Mosses' Public Inaugural Lecture marking his recent appointment to a professorship at Swansea University. This Lecture, furthermore, was advertised and presented as the *BCS-FACS Keynote Lecture in Formal Methods*.

A highlight of the meeting this year was a lengthy discussion, led by Samson Abramsky and chaired by Faron Moller, on the formation of a Learned Society for Computer Science in the UK. Such a Society was first proposed for Theoretical Computer Science 18 months earlier in a widely-circulated letter signed by Samson Abramsky, Faron Moller and David Pym which received overwhelming support; this led directly to wider interest in the UK in creating a Learned Society which would envelop the whole of Computer Science. This mammoth endeavour is being developed by a Working Party involving representatives from all relevant national organizations, and the meeting demonstrated near-unanimous support for its efforts (with only a small handful disappointed that the idea of a UK Society just for TCS would now be realized only as a sub-group of a wider organization).

BCTCS 2007 will be hosted jointly by Oxford and Oxford Brookes Universities in Oxford University's St. Anne's College from **2 – 5 April 2007**. Researchers and PhD students wishing to contribute talks concerning any aspect of theoretical computer science are warmly welcomed to do so. Further details are available from the BCTCS website [http://www.bctcs.ac.uk] .

FACS FACTS Issue 2006-3

Call for Submissions

Deadline **30 November 2006**

We welcome contributions for the next issue of *FACS FACTS*, in particular:

- Letters to the Editor
- Conference reports
- Reports on funded projects and initiatives
- Calls for papers
- Workshop announcements
- Seminar announcements
- Formal methods websites of interest
- Abstracts of PhD theses in the formal methods area
- Formal methods anecdotes
- Formal methods activities around the world
- Formal methods success stories
- News from formal methods-related organizations
- Experiences of using formal methods tools
- Novel applications of formal methods
- Technical articles
- Tutorials
- Book announcements
- Book reviews
- Adverts for upcoming conferences
- Job adverts
- Puzzles and light-hearted items

Please send your submissions (in Microsoft Word, LaTeX or plain text) to Paul Boca [editor@facsfacts.info], the Newsletter Editor, by **30 November 2006**.

If you would like to be an official *FACS FACTS* reporter or a guest columnist, please contact the Editor.

Perspectives on Formal Methods in the Last 25 Years

John Fitzgerald[1] (Editor)

Contributors: J-R. Abrial[2] & I.H. Hayes[3] & Cliff B. Jones[4] & John V. Tucker[5]
Transcription: Ivana Mijajlovic[6] & Ken Pierce[7]

Contents

John
Tucker

Cliff
Jones

Ian
Hayes

Jean-Raymond
Abrial

[1] University of Newcastle upon Tyne
[2] ETH Zürich
[3] ARC Centre for Complex Systems and School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia
[4] University of Newscastle upon Tyne
[5] Univesity of Wales Swansea
[6] Queen Mary, University of London
[7] University of Newcastle upon Tyne

1 Introduction

Mathematically-based "formal" methods for developing software and systems have had an interesting history. Over the past twenty-five years, the subject has moved from controversies surrounding code verification, through work on data types, design methodology, refinement and "Lightweight" Formal Methods, to automated proof and model-checking technology. The panel discussion recorded here brought together four computer scientists who have been active as leading researchers and practitioners in the field over the last quarter century. Held at the BCS London Headquarters, Southampton Street, on 30th January 2006, it provided an opportunity to learn about the motivations behind some of the major developments in the field, to discuss trends, fashions, successes and failures and set them in their recent historical context.

The panelists were Jean-Raymond Abrial of ETH Zürich, Ian Hayes from the University of Queensland, Cliff Jones, now at the University of Newcastle upon Tyne, and John Tucker of Swansea University. John Fitzgerald, Chairman of Formal Methods Europe, chaired the meeting.

In forming the panel, we were not trying to provide comprehensive coverage of the range of formalisms that have been explored over the last twenty-five years. Instead, we chose to focus on strands of work related to model-based and property-oriented specification. Three of the panelists, Jean-Raymond Abrial, Ian Hayes and Cliff Jones, shared a common point of reference through their time working at Oxford in the late 1970s and early 1980s. We asked John Tucker to join the panel as a figure closely associated with property-oriented view of specification. Indeed the distinctions between the two approaches were a prominent subject of discussion.

The meeting was divided into two phases. Each panelist gave a short initial presentation, identifying issues that could be taken up in discussion in the second session. Three of the panelists have provided summaries of their presentations, given in Section 2 below. Topics identified in the presentations formed the starting points for a discussion among the panelists and between the panelists and audience. Section 3 is a slightly edited transcript based on an audio recording made during the meeting.

Although intended as a "history" meeting, the issues raised by the panelists were clearly still contemporary concerns and excited passionate responses. The main technical topics covered were the distinction between model-oriented and property-oriented specification, and the debate over logics handling partial functions. The recognition and perception of formal methods was a major topic, in particular the observation that formal analysis, especially once automated, "disappears" into tools supporting design or compilation, and loses the potential to be acknowledged as a formal methods success. Issues in education and training were also highlighted, notably the debate over whether the fundamentals of the discipline, notably in syntax, grammars and programming language semantics, should be core to the teaching of computing.

Records of the meeting, including slides, will be available at the Formal Methods Europe web site [`www.fmeurope.org`] and the BCS-FACS website [`www.bcs-facs.org`]. Corrections and clarifications should be directed to John Fitzgerald [`John.Fitzgerald@ncl.ac.uk`].

Acknowledgments The editor is particularly grateful to Ken Pierce and Ivana Mijajlovic for their time and effort in producing a coherent transcript of an often lively discussion, using an audio recording prepared by Conrad Taylor. We are grateful to the British Computer Society, its staff and members, for the provision of the venue; to The Centre for Software Reliability, Formal Methods Europe and BCS-FACS for generous financial support, and to Paul Boca, BCS-FACS Secretary, for organizing the event.

2 Position Statements

2.1 John V Tucker: History of Abstract Data Types and their Specification

In this talk I described the origin of the modern conception of data types in programming languages and programming methodology, and the development of the algebraic theory of data based on many sorted algebras, equations and term rewriting. The period was 1966 to 1985, from A Wijngaarden on axiomatising numerical data types to my work with Jan Bergstra on computable data types and the widespread application of algebraic methods.

I considered the programming methodology literature, as represented by IFIP WG 2.3. E W Dijkstra's interest in specification started with his writing programs from specifications of un-built machines at the MC (now CWI) in 1950s. The emphasis on structured programming, specification, and reasoning was influential in making software a subject for theoretical analysis. The treatment of data in C A R Hoare's axiomatic approach to language definition of 1969, and in data refinement of 1972, helped untie data types from code. David Parnas work on software engineering in Philips Apeldoorn in 1971 led to his ideas on information hiding, the fundamental role of interfaces, and documentation, which freed data via the notion of module.

S Zilles made an independent formal study of data types in 1974-77; he knew about axioms and presentations and the Birkoff-Lipson paper on heterogenous (= many sorted) universal algebras of 1970. Combined with Barbara Liskov's introduction of modules as a collection of procedures with information hiding, in Venus (1972) and the CLU language (1976), a complete treatment of data types became possible. At the same time, there was a study of the representation independent specification of data in J V Guttag's PhD Thesis in 1975. But the subject took its current mathematical form through the work of the ADJ Group: Jim Thatcher, Eric Wagner, Jesse B Wright and Calvin Elgot at IBM Yorktown Heights, and Joseph Goguen, who wrote many basic papers on equational specifications, initial algebras, parameterisation, errors, etc., starting 1975.

I also noted the early work of Peter Landin, Rod Burstall, and Tom Maibaum on using algebraic methods for general questions about program languages; and the huge CIP project on software engineering.

Finally, I came to my work with Jan Bergstra on the classification of power of algebraic specifications using initial, final semantics and complete term rewriting systems. By 1985, module algebra, process algebra and tools such as ASF-SDF were in production in Holland, entirely dependent on the algebraic theory of the previous 10 years. I was delighted to pay tribute to some of the many intellectual pleasures of hanging out in Leiden, Utrecht and Amsterdam from 1979 to the present.

2.2 Cliff B Jones: A few threads around VDM

There are in fact many "threads" around the early days of VDM (I'll spell out the acronym in a while). There is the whole background of work on the verification of programs which slowly moved in the direction of methods for getting programs "right by construction". My version of this story is written up in [Jon03]. The move from *post facto* verification to using broadly the same ideas in design was, in my opinion, absolutely crucial to the usefulness of "formal methods". The name "VDM" was really a play on "VDL" and betrays the fact that one influence on VDM was the research on formally describing the semantics of programming languages. During the 1960s, the IBM Laboratory in Vienna wrote three major versions of an operational semantics for the PL/I programming language. Internally, this work was called "ULD-III" (the third version of a "Uniform Language Description") but the notation used was known externally as the "Vienna Definition Language". (The authors of VDL always acknowledged John McCarthy, Peter Landin and Cal Elgot for inspiration — see [LW69, BW71, JL71, Luc81].)[1]

[1] Another book on VDL was [Oll75]. I met Alex in Vienna but only learnt from John Tuckers' talk at this event that he had played a role in influencing the research in The Netherlands.

Although I'll mention below other work before 1973, the name "VDM" came about during the attempt to develop formally a PL/I compiler for a novel IBM machine (which never became a product). The group which worked on denotational semantics at the Vienna Lab from 1973-75 spoke of the "Vienna Development Method" (notice the change in the expansion of the "D"). The language definition work is described in [BBH^{+}74, BJ78, BJ82] and a discussion of the move from VDL to VDM is contained in [Jon01] but this is not where I want to focus this description.[2] Nor do I really have the space/time to discuss –what is to me– the fascinating challenges relating to concurrency (see [dR01]).[3]

One thing that connects with John Tucker's talk is that the Vienna group did use – what I would call – "property-oriented specifications". In fact, the first time I saw anyone try to capture the meaning of the operations on a "stack" by giving properties was in an unpublished paper by Lucas and Walk given to a group of patent lawyers in the 1960s.[4] One can see why this particular extension of McCarthy's basic "abstract interpreter" idea was needed by looking at the model of PL/I storage in the VDL descriptions — see [BW71].

The thread on which I really want to focus is that of VDM as a development method for general (i.e. not just compilers) programs. I feel the work on "data reification" was important and the story leading up to the happy time I spent sharing an office in Oxford with my good friend Jean-Raymond is fun.

I had spent a first two year stint at IBM's Vienna Lab starting August 1968. The main work on the VDL descriptions was already done and I went to look at whether they could be used to avoid the mess I had just seen in IBM's Hursley Lab which might be described as "trying to test quality into an imperfect design".

The two years were enormous fun and I learnt a great deal.[5] Partly, we found out how excesses of "operational semantics" (e.g. the "Grand State") made compiler proofs more difficult.

Of special importance was the work on proving properties of machines working on different data types. (Recall that VDL already used sets, sequences and "composite objects".) Lucas had used a "twin machine" (with what we would now call a "data type invariant" linking the two representations) in [Luc68].[6] I realised that in nearly all cases one could work with a "retrieve function" (homomorphism) from the (more) concrete to the abstract representation [Jon70].

Back in Hursley, 1970-72, I managed a small "Advanced Technology" group. Apart from experimenting with a style of definition which avoided some of the problems Lucas and I had found [JL71] with operational semantics (a "functional semantics" of Algol 60 is given in [ACJ72]), I began to look more seriously at developing "general programs". Reports like [Jon72b, Jon73] used post conditions which specified a relation(this was to become a distinctive aspect of VDM — at that time, most researchers followed Hoare's use of post-conditions which were predicates of just the final state (with various fudges to talk about constants for initial values) and a development of Earley's Recogniser [Jon72a] made another experiment with data refinement (interestingly, a more "property oriented" approach).

Peter Lucas called me in late 1972 and said that the Lab had been given a project to develop a PL/I compiler using formal methods (of its choice) — I think I accepted his invitation to return even before it was made! We also had the chance to pull in some new people and Dines Bjørner was hired from IBM's San Jose Lab. That second spell in Vienna yielded the language description part of VDM: notably the denotational description [BBH^{+}74] was a fraction of the size of the VDL descriptions (often, though unkindly, referred to as the "Vienna telephone directories"). But the machine for which we were designing the compiler was "killed" and the material on (the language

[2]Of more relevance to the discussion here is [Jon99].

[3]There is also an interesting story to be told about "support tools" — I was one of Jim King's first customers for "Effigy" [Kin69, Kin71].

[4]I later learned about the PhD research of John Guttag [Gut75] from his supervisor Jim Horning at a WG 2.3 meeting; Steve Zilles [Zil74] was about the same time. It was a challenge from these folk to say why "model oriented" specifications did not possess "implementation bias" which led to [Jon77] (which was thought out at the Niagara-on-the-Lake WG 2.3 meeting).

[5]Dana Scott's 1969 vist to the Lab was memorable! He came clutching the hand written manuscript of [dBS69].

[6]It was a side effect of working in an industrial lab that much of the interesting research was only "published" as Technical Reports/Notes.

semantics part of) VDM almost never saw the light of day. The checkpoint of [BJ78] was extremely important in the preservation of the research because the group had dispersed around the globe.

My next place of work turned out to be IBM's "European Systems Research Institute" at La Hulpe near Brussels. There I taught experienced IBM engineers about the idea of formally developing programs and wrote what was to be the first book (in Tony Hoare's famous "red and white" series for Prentice Hall) on this aspect of VDM [Jon80].[7] One of the things I am proud of is that I took "data refinement" seriously at this time; it was years before other books did so and there are still books published on "formal methods" for program design that only talk about program combinators.

During the time in Belgium, I received two invitations to "regularise my resume" (the wording comes from Brian Randell's invitation which was not the one I eventually accepted). I had become interested in computing while at Grammar School and had skipped the conventional "first" degree — there weren't any in computing at that time. So I ended up doing a doctorate under Tony Hoare's supervision in Oxford from 1979-81 (the ideas of rely/guarantee conditions date from this time — but I said I'd leave the concurrency story for another time).

Tony wrote to me (and I'd guess to Jean-Raymond) suggesting that our independent research on ways of specifying and reasoning about programs had a lot in common and that it would be fun to discuss it. Tony "facilitated" this by putting us in the same office in 45 Banbury Road.[8]

Abrial and I (and many other friends from this time) certainly did have fun discussing aspects of specification and design. There was never a problem with details of notation: Jean-Raymond would write Z-like notation on one half of the board and I would scribble VDM on the other side. We all knew it was the deeper concepts which were crucial.

Bear in mind that I considered VDM's style of sequential program development was reasonably thought through (and that I was supposed to be doing my DPhil research on concurrency). There was one problem that we could all see was unsolved: in order to build large specifications one might want to use specifications of smaller constituent parts. The need to (in the most general sense) "promote" an operation on an object to work within a larger state was often discussed.[9] This is a very good topic to explore today with this super line up of panelists.

I concede at once that the "operation quotation" idea in VDM [Jon86] was "clunky"; but I also have to say that I never felt that (what became) Z's "schema calculus" was safe in the hands of other than real experts. We would never design a programming language with such a textual capture concept; why assume it was a good idea for specification languages?[10]

Contrary to assumptions by some people who were not there, I was far from negative about Z. In fact, I consistently encouraged them to publish Z[11] and was involved in getting the IBM/CICS project for Oxford. I happen to think that Ian Hayes' [Hay86] (first and second editions) is one of the best books around on "formal modelling". Ian and I went on both to propose changes to VDM and Z that would remove some of the irrelevant differences and to write our infamous "Magic Roundabout" paper [HJN94] (a reprint of this paper appears in the current issue of *FACS FACTS*, pages 56–77).

I'd like to add a few words about VDM post Oxford. When I arrived in Manchester, Peter Aczel had been studying [Jon80]. He wrote me a letter [Acz82] (which he sadly saw as not worth publishing) about Hoare axioms and my proof rules. His comments indicated that Jones was obviously right to choose post-conditions which were relations (of initial and final states) but his rules are "unmemorable" (a less polite person might have said ugly!). Peter's proposed cleaning up of the style for presenting inference rules for relational post conditions yields rules which are as elegant and memorable as the Floyd/Hoare rules. (And in my opinion, the well-founded relation has always been more natural than Dijkstra's "variant function" [DS90].) The clarity of these

[7]There is an amusing side show on getting the book printed on an APS/5 laser printer in San Jose.

[8]Another very important visitor was Lockwood Morris. We together "exhumed" Turing's early verification paper [Tur49, MJ84] and Lockwood helped me see that the way invariants were handled in [Jon80] made proofs more difficult than they need be — this led directly to the change to viewing invariants as type restrictions in [Jon86].

[9]As far as I can remember, Tim Clement had the best proposal at the time.

[10]Martin Henson was in the audience and has very interesting new proposals.

[11]I took to the meeting a "spoof" paper with Bernard Sufrin's name on the cover: "Reading Specifications" was a rather crude prompt!

rules was one of the major stimuli to writing [Jon86] in which I also took the plunge and presented "data reification" *before* "operation decomposition".

VDM's influence on other notations/methods has been considerable:[12]

- Jim Horning acknowledged the influence on Larch [GHW85]
- VVSL [Mid93] is a close derivative of VDM
- RAISE [Gro95, Gro92]
- I am honoured by Jean-Raymond's generous acknowledgement to the influence on B [Abr96]
- Our esteemed chairman has shown how to combine VDM with OO concepts in [FL98, FLM$^+$05]

Discussion points

We were invited to seed the discussion with some "provocative" comments of our own. I offered:

- VDM, Z and B (as specification languages) are close cousins; they all use a "model oriented" approach; the approach of documenting "abstract models" has proved very productive. (But it is fun to look at the differences between cousins.)
- one of the most interesting differences is the way in which specifications of components are used in the specifications of larger systems;
- in the 1970s/80s most researchers (outside of the VDM/Z schools) were working on "property oriented" specification techniques; the model-orinted camp felt in a distinct minority (thus the "bias" test);
- data refinement is more important than proof rules for programming constructs;
- formalism pays off — use in the early stages of design
- (ISO) standards don't (pay off)!

2.3 Ian Hayes

My introduction to Oxford and Z (1982). In January of 1982 I was visiting Carroll Morgan at Oxford and met amongst others Tony Hoare, Ib Holm Sørensen, and Bernard Sufrin. During the visit, as well as being exposed to the cold and grey of a Oxford winter for the first time, I was exposed to the pre-"schema calculus" form of Z via Jean-Raymond Abrial's handbook on Z, and Bernard's editor specification, and I went away with a pile of reading including more of Jean-Raymond's writings and Cliff Jones's *Software Development: A Rigorous Approach.*

The IBM CICS Project (1983-85). A year later, in another freezing cold January, I returned to Oxford to work as a researcher on the IBM CICS project, which was tackling the challenge of formalising the CICS Application Programmer's Interface using Z. By now Z had grown to include the early schema calculus as presented by Carroll Morgan and Bernard Sufrin in their Unix Filing System paper (IEEE Transactions on Software Engineering).

The project had Ib Holm Sørensen and myself as researchers, working under the direction of Tony Hoare. We also had Cliff Jones and Rod Burstall as consultants. We worked with the likes of Peter Collins and Peter Lupton at IBM, spending a day a week at the IBM (UK) Laboratories at Hursley.

[12] One could also mention the many (at first) "VDM-Europe" symposia [BJMN87]; later broadened to "FM-(E)" of which our panel chairman is also chairman.

Motivations. The researchers were primarily programmers, rather than theoreticians, and the emphasis was on trying to specify real systems. Our objective was to devise a specification of the system that was an accurate description of its desired behaviour but also as easy to comprehend as possible. To this end we emphasised using set theory to provide abstract descriptions of the system's state, implicit specification of operations using "relational" predicates, and building/structuring specifications from components. Even then we were interested in combining multiple views of a system to give a clear specification.

Early Z. In the early days, operations were specified in Z as either functions or relations. Schemas were initially just used to specify a "record" type, but with an invariant constraining the permissible combinations. They weren't used to specify operations, but they were used to specify the state of the system. As well as being used for records, they started being used as a shorthand in quantifiers. For example,

$$\forall\, s : \mathrm{seq}\,\mathbb{N};\ n : \mathbb{N}_1 \bullet n \leq \#s \Rightarrow ...$$

was converted to

$$\forall\, s : \mathrm{seq}\,\mathbb{N};\ n : \mathbb{N}_1 \mid n \leq \#s \bullet ...$$

and abbreviated to

$$\forall\, SS \bullet ...$$

where

$$\begin{array}{|l}
\hline
SS \\
s : \mathrm{seq}\,\mathbb{N} \\
n : \mathbb{N}_1 \\
\hline
n \leq \#s \\
\hline
\end{array}$$

Operations were specified as functions, but using schemas for the state:

$$\begin{array}{l}
\lambda\, x : \mathbb{N} \bullet \\
\qquad (\lambda\, SS \bullet (\mu\, SS' \mid s' = s \frown \langle x \rangle \wedge n' = n))
\end{array}$$

This allowed succinct reference to system state as well as implicit inclusion of the state invariant as the predicate part of the schema describing the state.

Using schemas to specify operations. The next step was to use schemas to give a relational specification of an operation, as first used in the Unix Filing System paper. This allowed specifications to be nondeterministic, and preconditions could be extracted by taking the domain of the relation. To allow the state of the system to be specified as a schema (along with its implicit state invariant), schema inclusion and schema decoration (initially with just a "′") were required. Then came ΔS and ΞS and variations on these to help manage the frame problem.

From there, schema disjunction was used to allow alternative behaviours (e.g. error behaviours) to be specified, and schema conjunction and hiding were used for promoting operations to a larger state space. Schema conjunction was of particular interest as it didn't have a programming language counterpart.

Other operations on schemas included relational composition which, although it was written with a ";", was quite different to the sequential composition in programming languages.

VDM and Z. The work at Oxford was heavily influenced by the pioneering work of Jean-Raymond on Z as well as by VDM. At one stage we tried to converge the syntax for the mathematical notation used in VDM and Z. This wasn't completely successful, but they are much closer than what they might have been. There are also great similarities between the refinement rules used in Z and VDM.

Refinement. Peter Lupton and Ib Holm Sørensen started to use Z schemas as components of programs written in Dijkstra's guarded command language. This influenced Carroll Morgan's development of his refinement calculus, which helped provide a detailed semantic basis for what Peter and Ib had been doing. But the history of the refinement calculus is another story, so I'll finish here.

3 Discussion

3.1 Panel reactions

John Fitzgerald: First of all I would like to thank all four of the panelists for giving such a broad range of talks trying to describe twenty five years in twelve minutes. I am going to first of all ask the panelists to respond to any of the issues that they felt came out of the other presentations and just really to identify the issues that we should discuss in the open session.

First of all I would like to ask John if there is anything that stood out for him from the other talks. Perhaps you feel the other talks were coming from the areas of the discipline that were rather closer to one another historically and intellectually.

John Tucker: Well perhaps in some sense, my own talk was rather narrowly focused. There are overlaps in different ways, but I would first like to invite Cliff to define very precisely the difference between "model approach" and "property approach".

I think what also came out is this idea of the way we handle these subjects educationally: how well known, for example, are the highlights of the activity in the last 25 years? It is a very big thing this thing called computer science and it is quite natural that all colleagues should have some information about what is achieved in the last 25 years, some primary landmarks, for example. Traveling on the metro is very important for formal methods, and we should do more of it because one can claim, "here is something where these things have been used." Of course there are many other examples but it is not the case that the average researcher in formal methods, never mind the average computer scientist, could recite some classic examples where these techniques have been used.

If I were looking in engineering, say, forty years ago, I would say that the finite element method – a very important modeling technique – would not be so well known among civil engineers. But if you remember the old problems of the box girder bridge that kept falling down, the entire analysis of how these things work and the explanations of this big problem were analyzed by these kinds of techniques. So gradually things like finite element methods appear on the horizon of the average computer scientist. So that is something I would like to put to the audience as much as the panelists.

John Fitzgerald: Good, thank you very much. Cliff are there any points you would like to raise?

Cliff Jones: I absolutely agree we should talk about technical distinction and education. Can I just pick up this recognition of formal methods, because I think it is a very difficult one. So, people from Praxis here will tell you when you land at Heathrow that formal methods matter there as well. In fact, a certain rather large VDM specification has come back to haunt me, because we are using it as one of the "drivers" for reformulation in Event-B within the EU-funded Rodin project, thanks to Praxis making it available to us, I have to look at this rather large VDM specification.

But I would like to pick up something Tom Maibaum has said many times and this is that it is the "curse of formal methods" is that as soon as something gets used it gets taken into "software engineering" and is no longer "formal methods"; they are not using "formal methods" because that is the bit they have not taken (yet). It is sort of a definitional Catch-22. Maybe we could talk a bit about the problem of the *perception* of formal methods and their impact, versus the reality.

John Fitzgerald: Thank you. Ian, are there any comments you would like to make?

Ian Hayes: I wanted to pick up on two issues. I guess education and application in industry.

From the Australian perspective, I don't see that we have progressed much in the last 25 years, especially in applications in industry. I think in the UK and Europe it's much better than in Australia. If anybody has got any clues for me on how to improve things, both in applications and in education, that would be interesting. Even convincing my colleagues is sometimes difficult.

John Fitzgerald: Thank you, Ian. Jean-Raymond?

Jean-Raymond Abrial: I have almost the same worries as Ian has explained. It is easier to convince students than to convince colleagues. I think that education is extremely important; I also think that the field of software engineering these days is still focusing too much on the problem of discussing programming languages and adding features to programming languages. In "Software Engineering", the most important word is *engineering*; I think as such it is not taught enough in education.

Of course we have UML, but as my colleagues agree, its semantics and definition are not at all well enough defined. Again and again, I would like to put the emphasis on education. From my experience with using formal methods in industry, it is not difficult for engineers to learn something like Z, B or VDM, it is not difficult, it is far easier to learn than C++.

John Fitzgerald: Thanks very much.

3.2 Models vs properties

John Fitzgerald: I've had a couple of questions in advance from members of the audience and I think we'll fold those into the discussion around some of the topics that have already been mentioned. First of all, I'd like to address some of the more technical aspects; I'm going to put my friend Cliff on the spot on "models vs. properties", if we could talk around that topic.

Cliff Jones: I believe what characterises property-oriented specifications is that you write *equations between the operators*. In what I call model-oriented specifications, you introduce a model and you define things in terms of changes to that model, rather than equations between the operators. That's the distinction I would like to make.[13]

John Fitzgerald: John, did you find that a satisfactory distinction?

John Tucker: You can easily see in the case of the stack and other examples, that the guessing of equations comes quickly and easily and is designed not to bother about models.

Cliff Jones: Can I interrupt with an anecdote? We all know the equations of stacks. I'd heard Jim Horning talk about John Guttag's thesis very early on. I remember approaching a number of distinguished computer scientists (I'd better leave names out of this) and saying, "Could you dash off the axioms for queue for me?". Most of them scratched their heads. One very distinguished computer scientist, with great confidence, went to the blackboard and wrote the axioms for a ... stack. It's *not that easy* to get just the right equations.

John Tucker: Well to write the equations for the stack *properly* is extremely difficult. I think I could go on about that ad nauseam. Largely because the stack is one of those things, if there is such a thing, that is a prima facie case where one might be advised to use partial functions.

However, putting aside that unique distinction that the stack has got, what I was trying to say was that with certain structures, like the stack, you start by guessing these operators and properties and of course you run into trouble. The property-oriented idea comes when you see the operators and you start wanting to write down these properties.

Typically, although this is plausible, it strikes me that for the average problem that you want to specify, you're rather clueless over the properties. So really you need to make models first in order see whether or not you want to have specifications that are somewhat more abstract, later. I think it's fair to say that many techniques in the algebraic specification method, when you actually get round to using them, do require you to make these models first and then find these axiomatic equations as a second stage.

Tom Maibaum: I think that the discussion so far perhaps misses the point, because really there is a simple distinction between property-oriented and model-oriented approaches. In the

[13]A fragment of the draft paper "Model-oriented Specifications" by Clement and Jones, comparing algebraic and model-based approaches, can be found in this issue of *FACS FACTS — see pages 39–53.*

property-oriented approach, you take some standard logic and you specify theories over that standard logic. In the model-oriented approach, you extend that logic with things like sets and you write theories over this extended logic. Otherwise I agree that the distinction is neither here nor there, if you accept both approaches as logics over which you write specifications.

Cliff Jones: I'm comfortable with that.

Jean-Raymond Abrial: I had to teach a course, some time ago, on programming languages. There exists this distinction between imperative languages and logic languages and so on; so the exercise that I set the students was to develop an interpreter for logic programming in Pascal and an interpreter for Pascal in Prolog. I believe that between models and properties we can do the same thing, we can simulate one in the other and that would be very interesting. It is possible to completely simulate property-oriented specifications in models and vice versa.

John Fitzgerald: What a diplomatic note! Are there any other comments or questions from the audience on this issue?

Peter Mosses: I felt somehow rather dissatisfied with Tom Maibaum's exposition; washing out the difference and saying that it's all a matter of logic and which logic you happen to prefer.

It seems to me that the model-oriented approach is a definitional approach. We give a definition for our various functions based on what, in an algebraic sense, would be hidden things (for example, the concrete representation in an abstract model of particular functions). Rather that just relating the things in the signature in the interface directly to each other, as in the property-oriented approach, which are essentially the things the user is concerned with. I feel this is an important distinction.

3.3 Partial Functions

John Fitzgerald: I'm going to ask Peter Mosses to stand up again. If models vs. properties was controversial, perhaps the next question will be just as good.

Peter Mosses: The different formal methods tend to have differences of nuance, or perhaps even bigger differences, in their treatment of partial functions. There was some lively debate about this around ten years ago. I'd like to ask the panelists what their perceptions are of that debate about the differences between the approaches to partial functions in the different frameworks. Did any kind of consensus emerge from that debate and what would they recommend now as the best way of doing it?

John Fitzgerald: I'm going to ask the panel to concentrate on the history and their recollections of the debate and less on holding the debate again, if possible.

Cliff Jones: The subject of logics for partial functions comes up in one of McCarthy's papers from the 1960s and I first met the topic when Peter Lucas introduced me to that paper. The Vienna Definition Language had indeed taken McCarthy's "ordered view" of operators: "false and undefined" evaluated to "false", but "undefined and false" was "undefined". This seemed an unfortunate property to me. I'm not going to go through all the attempts I made to circumvent this problem since,[14] but I met that problem during my first stay in Vienna during the late 60s and I didn't like the answer.

I subsequently attempted to use bounded quantifiers and I also attempted to use two sets of operators (as Dijkstra does) called AND/CAND, where commutativity is preserved when possible and the ordered version is used as necessary; but the laws for this approach are horrible. The only approach I never tried was the one that Z used at one time, where the factorial of -1 denotes some unknown integer — I really don't like that approach.

The cute answer would be that after I came up with the "Logic of Partial Functions" with Jen Cheng and Howard Barringer, I saw no need to try further.

The honest answer would be that the differences, which I agree are still there, come from what you're trying to do. If you look very carefully at the formal proofs that we did in VDM, the way we handle partial functions works rater neatly. I was talking to Jim Woodcock about this last

[14]See [Jon06] for references.

week and his point is that the Z proofs are just different proofs; they don't come up with the same sort of constructs and therefore they suggest different requirements for the logic.

I believe there is no consensus and I doubt there ever will be, because the differences probably come more from what you are doing with the logic than just cussedness or blindness.

John Fitzgerald: Ian Hayes, do you agree there's no consensus?

Ian Hayes: I was just going to make the comment that when I'm presenting this to my students, one of the things that I've learned over the years is that I need to distinguish between undefinedness of expressions in a programming languages and those in specifications. It is often the case that exactly the same syntax is used and people get confused about the two different issues. In a programming language, things like CAND make sense, but in a specification they don't.

John Fitzgerald: Any comments from the many people in the audience who have been involved in this issue?

Bernie Cohen: Undefinedness in a specification has got be be related to an absence of specificity in the statement of demand for the product. So it's usually plausible to generalize and say that in this particular circumstance, one resolves the undefinedness this way or that way. It depends on how the system is going to be used and that's a question of the demand of the user. That is a major gap in the whole development of formal methods and I think that's a partial answer to Jean-Raymond's prompt about what is missing from Event-B, in that we've never learned how to deal with demand and that's been absent from our formal methods for the last 25 years.

Jean-Raymond Abrial: I think this is a very old mathematical problem. Mathematicians have always made progress in mathematics by saying, "by the way, it is possible to talk about division by zero or complex solutions to equations", for example. A normal mathematician will, when working with series, for example, be sure that the series is not divergent.

My view is that proving that the problem you have has not got anything that is undefined or ill-defined is (along with type checking), another filter in proving the well-definedness of your expression. Lexical analysis, syntactic analysis, type-checking, well-definedness and then proof. If you go through all these filters, then at some point you are certain that your specification is well-defined; you factorize out completely the use of division by zero or the maximum of an empty set and so on. This is my view, I know that Cliff and I do not share the same view, but we are extremely good friends and we still argue about it!

Rob Arthan: My view on this topic is that you get out from a logic or a proof something that is proportional to the amount of work you put into it. So if you do a constructed proof you get something really good out, you get a program.

If you do a logic of partial functions proof, you get a guarantee that your specification doesn't depend on the result of two undefined values. If you do a really classical proof, in which 1 / 0 is some unknown integer, then what you often get is a proof, because you've made life easy for yourself. I think it's very much a technological question and a question about what you want. If you want good tool support and lots of automation then maybe you need classical reasoning; perhaps in 20 years time our automation will be so much better in that we can all do logic of partial functions using tools.

Cliff Jones: I'm glad you brought tools into the discussion. If you've got the right sort of mechanical theorem assisting tool, then you really don't notice the difference between the logics, you just learn to live within it. Frankly today, with the possible exception of Click and Prove, most of the tools are so difficult to use, that the least of your problems is the specific axiom set.

That, I see as an opportunity as opposed to a reason to give up; we need to do a lot more work on making our proof tools more usable. The proof tool which is good and embeds a particular logic will define which logic we're using in ten years time, I think.

Which is a worrying observation.

John Tucker: I have to say a few words on partial functions. If you work with the so-called algebraic specification method, at the end of the day, you are making a statement that says, I'm going to have operators and formulae expressing properties of these operators. So you've just got these operators in the signature and equations (possibly conditional equations) and not much more, to play with.

As I mentioned in the presentation, this can express virtually anything you want, which is a testimony to how complicated equations are. Logically, you are dealing with a very simple logic; you write down terms and you use the equations to rewrite them. Everything is stripped down to nothing. If you let these operators be partial, suddenly you see that a really big, nasty set of problems start emerging in these calculae.

Even in the simple case of equations with operators, partial functions start stretching the mind and patience of those who work with algebraic methods. I would say, that this is the sort of experience that you want to be very careful about using. On the other hand, when you use algebraic methods, in terms of the great spectrum of formal methods, you really are using a high-resolution microscope. You are looking at very sharp, precise details of a model or a set of models, that satisfy the axioms — increasing the technical rigour.

I agree with the idea of avoiding partial operations, try to make these models in situations or with conditions where you are able to check that the partial operations don't appear. But on those occasions where you want a wider view, algebraic specifications show up the great gap there is in technical terms between a world without them and a world with them.

If you deal with the stack, it turns out that it is a non-trivial problem to formulate any kind of satisfactory specification. You can do it, but it's difficult. Whereas in other situations, such as division by zero, if you want to have the inverse of zero equals zero, go ahead. This produces the most fantastic calculus with everything you could possibly want, just don't use it for continuous data types, because it does interfere with the continuity — but not the algebra.

3.4 Recognition and perception of formal methods

John Fitzgerald: I would like to broaden the discussion a little to some grander themes that came out of the presentations and the comments afterwards. If we could concentrate for a while on "recognition of formal methods" — the perception of formal methods, in some sort of engineering community.

I'd like to ask the panelists how they feel that perception has changed, if at all, over the last 25 years. Is it better, or is it worse, and why? Would any of the panelists like to start off? Jean-Raymond?

Jean-Raymond Abrial: I think that in some sense it has not changed, but in ways it has. For example, at Southampton University, the chair of Michael Butler is called "Formal Methods". On the other hand, at Ecole Polytechnique in France, they do not recognize, at all, this idea of formal methods. In Zurich, where I am presently, there is a very strong "Chair of Software Engineering", where they do almost no formal methods at all, they just add new language features and things like this.

So the situation is quite different from one place to another; but again I think the problem is the one of tools. If we have good tools for formal methods we can teach them together with the tool, and I think that is very, very important. I remember, for example, Hoare logic was taught years ago and then in many universities it was abandoned, because there were not enough tools around for Hoare logic. This is less and less the case now, because there are more and more things developing in this area.

So the situation is not simple, the situation is complex, but I think from the point of view of the formal methodists, we have to continue, it is very important.

Another interesting example is that of artificial intelligence. I think in formal methods we go slowly, while in artificial intelligence there was a peak and then it went down. I think we have to continue to go at that speed.

John Fitzgerald: Thank you. Are there any other comments from the members of the panel?

Brian Wichmann: Cliff mentioned CICS which was a mega user system. What systems do we have these days, which have been developed using formal methods, even specified using formal methods, which are in the same category?

Cliff Jones: Brian's question is very interesting, but it comes back to Tom's puzzle of formal methods. Microsoft are now making extensive use of verification style technologies getting the many bugs out of their software. That's not called –in general– formal methods. There is a wonderful quote from Bill Gates which talks about the adoption of formal methods. It is a speech in 2002 or in 2003 –in front of a very wide audience– how they are beginning to adopt formal methods in their driver testing-debugging cycle.

There is no bigger software example than that, but I suspect it is not seen as a formal methods success. They need assertions in order to cut the false alarms in their extended static checking. Where did assertions come from? So, I think there are plenty of examples.

Richard Bornat: I will correct you on that, Cliff. It is seen as a formal methods success, it's based on Byron Cook's tool, which is SLAM, and it is absolutely formal methods success.

But I think the interesting point is that of the question of what is the status of formal methods. We are having this discussion in England, and two people on the panel, one of them is French, the other is half Dutch, and one is Australian, and the fact is, software engineering, it's important to be as rude as we possibly can about software engineering. I will mix this up with the discussion about education.

Dijkstra defined software engineering as how programming should be taught. It is a process which has been developed from the English class system. A question they ask over and over again in their histories, how do we run a system where upper class idiots are supervising lower-class clever clogs? Over and over again: the first World War was a disaster, the English Civil War was a disaster, over and over again they get the answer, it does not work.

The fact is software engineering is nothing to do with computer science, never has been and never will be. When they steal things from us, they just steal things from us and they don't know how to use them.

Part of the problem we have at the moment is, there is a massive failure of confidence in computer science education and it is caused by the fact that we are not sure, why it is that most of the people who come to the computer science departments, can never learn programming. Interestingly enough, this is an advert for me. A student and I have discovered a test to tell you who in your intake can never learn to program – and I mean never – and who in your intake can learn to program.

Half of those who can learn to program will not like it and they will become software engineers. We should, as formal methodists, recognize that 60% of the male population can never learn to program and I am not exaggerating. It is a big problem because that means we have only got 20% who could be software engineers and 20% who could be programmers.

That is actually if we can begin to sort them out and we can continue to be extremely rude about software engineering.

We also have to be rude about formal methods, because the truth is there are not that very many programs that we care about enough to have Jean-Raymond Abrial spend 5 years writing them. Most of the times someone says, "I would like you to order my chickens in order of who lays the most eggs." It does not matter if your program crashes most of the week; formal methods is completely inappropriate here. You want to write the program today and run it tomorrow. You do not want to sit down for 5 years and get it right. It's not worth it! The chickens will be dead!

John Fitzgerald: Thank you. I would like to keep the education point until a bit later, and for the moment stick with the perception of formal methods in the profession.

Peter Amey: I think there are several ingredients. I think the biggest problem is having the name "Formal Methods" with a capital F and a capital M with big quotes around it. We have heard about things going into tools, we have heard about industry adoption and documenting using formal methods.

I think actually formal methods will have succeeded when it does vanish, because people are just doing this because it is good engineering and they are doing it because it is embodied in the tools that they are using.

We have got a whole lot of Spark users around the world putting in proofs all over the place. None of them would admit to using "Formal Methods" and probably wouldn't even understand what you meant if you said "Formal Methods". The secret is to make this attractive, usable engineering and to get rid of the capital letters; stopping it being something special and scary that everybody is frightened of because it has got a label.

John Fitzgerald: Thank you very much. Any other comments from the audience, please?

Martin Campbell-Kelly: The latest paradigm for producing quality software is the "open source" initiative. I just want to know how those two things fit together — formal methods and open source — where "bugs become shallow under many eyes".

John Fitzgerald: Thank you. Any comments from the panel?

Jean-Raymond Abrial: I think open-source is very important for the tools and for the people using the tools. For example (this is a case that Cliff and I know of), in our RODIN project, one of the main emphases on the project was to have tools that are completely open-source. So I think it is very, very important to have this. There have been too many tools that are proprietary and that has not given any good results.

Cliff Jones: I am delighted Martin is here, because he is real historian. He helped me when I tried to write the history of program verification for the "Annals of Computer History" some years ago. I think he is too much of a historian that even open source is not that latest thing any more! I thought that you were going to say "XP".

What I really wanted to say constructively was that I would love to see our community –with or without capital F and M– publish public, open-source *specifications*. A lot of my career has been in industry, there was the time when I was relying on something called CORBA and I would loved to have had a formal specification up there that I could have just referred to and found out what this wretched thing was supposed to do — none of the implementations did the same!

I think if we could infiltrate people's working life by publishing a lot of, what Peter Mosses calls, formal descriptions (they are not necessarily "specifications" — specification has a formal legalistic status). But for things that people use, if they found it was handy to go off and look at this formal description (with the small f), that might influence what we are doing more than anything.

Jean-Raymond Abrial: One of the problems was with using formal methods in industry, researching the formal methods in industry, is that very, very often, what is lacking or what is very bad in industry is the requirements documents. So it has nothing to do with formal methods. Most of the time either it is extremely verbose or it is almost nothing and it has to be rewritten. So one of the main points, before formal methods (and this is again something to do with education), we have to educate young students about just writing good requirements documents.

Rob Arthan: On the take-up of formal methods, I started using formal methods in industry with VDM in the late 80s and Z ever since. One of the things that made life hard then was precisely something Cliff raised, which is the lack of standardization. Of course standardization was hell, I spent 12 years or more of my life on a Z standard committee and it was hell.

I hope things are going to get better and that we'll have got better at coming up with formal methods which are immediately accessible with tools and rigorous definitions of the languages themselves.

Cliff Jones: Well, I am really tempted to raise the question of how much harm we did by having different methods, which to a certain extent, as Ian and I showed, were different in completely silly ways — which symbol we chose to write something, for example.

Some differences are interesting, I believe they are technically interesting and need thinking about, but a lot of confusion was avoidable. A really classic problem was that Dines Bjørner put up a project to EU called VDM-Fit, which was "clean up VDM". That got turned down as far too boring, but he put up a project called RAISE, which was to design yet another specification language (as though we needed one) and the EU funded that!

Maybe we do need a way of sorting out some problems behind the scenes and putting out something that looks more uniform to the rest of the world, so that they don't get confused.

Bernie Cohen: On the subject of descriptions, providing descriptions in a formal language does not help, because they are always descriptions of what somebody else wants and not something that you want.

A very clear example of that comes from way back in the 70s from the HOS girls, who produced the world's first properly tool-supported formal method, with abstract data types and everything. They had nice contract from US to specify weapons acquisitions, up to, but not including, transition to war. One of the things they had to specify was a truck and they had a whole catalogue of abstract data types available and they went looking in it and they couldn't find "truck". So they started from scratch and they specified truck, and found out to they horror, after a couple of weeks of work, what they had was isomorphic to stack. They did not not know that in advance! Well it is obvious, a truck is something where what ever you put on first gets takes off last. That's the problem, you publish the specifications and the descriptions of things expressed in formal terms, but they are not the things that you want, they never are.

Tom Maibaum: I would just like to report that my impression is that this is a pessimistic discussion and I want to report something very optimistic. I want to report we have entered popular culture. So we are now in a position like physicists and mathematicians, to pull the wool over politician's eyes and say how what we are doing is great, and get a lots of funding.

How many of you have heard of the Fiver? Not one of you? Well the Fiver is an electronic bulletin sent out at five o'clock every day by the Guardian, via e-mail, with a humorous take on the day's events in football. So two or three weeks ago, when the draw for the world cup was taking place, they had a report about the ranking of the various teams participating and how this ranking was obtained. The description began by saying that in order to understand this complicated method, you needed to appreciate something about Iron Maiden and something else and a Ph.D. in the polymorphic Pi-calculus[15]. My contention is that we've made it. We don't have to worry any more!

Richard Bornat: For a large part of computer science, I would say the only interesting bits of computer science are the formal methods — and the bits of formal computer science that work are all in compilers.

Compilers do completely magical things. John Reynolds once told me that when someone told him that you would write a description of a program and a computer would write the program –this was before FORTRAN was invented – he thought it was magic. He couldn't see how that could possibly happen.

Now, of course, we know that it is not magic at all. When I sat behind Rod Burstall, he showed me how a computer could work out the type of an expression in ML. I thought that was magic — and I still think it is magic, even though I know how it works.

Formal methods is just bits of computer science that we haven't got into compilers yet. This suggests to us that the old bits of formal methods, interesting though they may be, are possibly not going to get into compilers at all. There are lots of new bits of formal methods, like Event B and the bits that I work on, which *are* going to get into compilers, aren't they?

So formal methods rolls on and it will always be that formal methods is never going to be invisible, because there are always be the stuff that we are trying to do, that we haven't done yet. 90% of it is not going to work, 10% of it is and and we don't know which 10% will. We go battling on with algebraic data types if you want to, VDM if you want to, Event B if you want to, Separation Logic if you want to. Formal methods is just what we haven't put in a compiler. As soon it gets into compiler it becomes invisible.

Brian Wichmann: Responding to Richard's point about compilers. Of course, there is a lot of that. But I am not really concerned so much with compilers, but what they produce. The fact of the matter at the moment is that (perhaps with the exception of Java if you don't use concurrency), almost every program that is ever written with more than a 1000 lines will, under some circumstance, execute unpredictably. I think that is, 50 years after the advent of the first compiler, atrocious. We should be disgusted with ourselves.

Cliff Jones: I think that our colleagues from Praxis will argue that there are several counter-examples (Spark).

Brian Wichmann: Yeah, but that's not 99% of computing, I'm afraid.

Cliff Jones: I agree, unfortunately. They would not mind if it was!

[15] "The Fifa Coca-Cola world rankings (yes, that's their official title) are so fiendishly complicated you need a PhD in polymorphic pi-calculus, a deep love of Iron Maiden, and a fondness for making jokes about "picking two from the top and one from the bottom, Carol" to understand them." Iron Maiden; and Slapstick Farce, Sean Ingle and Georgina Turner, The Fiver, Tuesday December 6, 2005.

3.5 Education

John Fitzgerald: I want to move on to the education issue, which several of you have mentioned and indeed two of the presentations did. By education I assume we mostly mean university education, but I'd like it if people talking about this issue could at least make a nod in the direction of industry training as well.

Peter Amey: One of the things that troubles me is the commercialization of university education, in that students are being asked to pay a lot more and they're coming out of universities in debt. As a result of that they seem to want to learn things that they believe have an immediate application and the universities are responding by giving them that. It seems to me that actually the universities are abandoning education and moving into training and actually they are quite distinct.

What I want from people who apply for jobs at Praxis to have, is an engineering education, upon which we can do technology specific training. I don't want people to come knowing what the latest programming language is, having no idea about engineering principles. I don't know how to solve that problem, but I think it's a big one and a growing one.

John Tucker: I drew attention to this in the narrow sense of data type theory, but generally speaking, I'm rather interested in what people have to say. We already have these problems about what is software engineering, what is formal and what is not formal. For example, Richard Bornat mentioned compilers, which are forever being expanded as programming languages get more and more abstract and high-level. As such the notion of a compiler grows, the specifications become more abstract and we all expect the machine to do the donkey work of producing simulations or translations into other formalisms.

If you take, for example, the 50s and 60s, when grammars started to appear in the vocabulary of everybody, people would learn a lot about grammars. When the tools came along, particularly with UNIX, many universities used to have plenty of interesting exercises for second year students. So if you take a fifteen year period, which some people here would probably be intimately connected with, you go from a situation where nobody could see how the future of compilation and higher-order languages would go, to second year students being able to do many compiler exercises.

So there's a question again there, is that formal methods? Well of course it is. It has come out of a formal community of people doing formal languages and working in connection to logic, yet it has become mainstream. Now ladies and gentlemen, show me your courses on grammars where you really go into detail on pumping lemmas and sophisticated algorithms. If you're not teaching the basic theory of context-free grammars to your students, what the hell are you doing? Grammars, abstract data types, finite state machines and various other things are part of a scientific analysis of the fundamental notions underlying software and computer science. We've gone so far as to forget about these things, which I find shocking.

Cliff Jones: It's even worse than you make out. I teach students semantics and want to assume they know about syntax; I couldn't figure out why they didn't until I looked into the Java books. We went through 40 Java books and not one of them had a concrete syntax for Java in it. The standard does, but of the normal textbooks that students use, very few of them have a grammar in them.

Richard Bornat: It's important never to be old. Ever since the beginning of time, the old have complained that the young are now rubbish. That they're not learning grammars anymore and so on. I wrote a book on compiling, which had lots of information on grammars in it and I'm extremely glad that we don't teach grammars anymore, they are not foundational they are accidental. Lots of people needed to know grammars, because lots of people needed to write compilers and compiler-like programs; now almost nobody does.

You can learn about grammars in about ten minutes by reading my book, it's extremely easy — grammars are not a problem. The old will always say the young are rubbish and the young will always be cleverer than you behind your back — just as we did, they are doing it to us now.

Cliff Jones: I disagree with several things. I never said the young were stupid, I think it's their teachers who are to blame!

Andy Krasan: I'm probably the oldest person here and I'm retired. I worked at Hursley on PL/I, on an interpreter that ran in 100k. My first piece of formal methods was when Harlan Mills walked through the door at Hursley and said the way to write a program is with structured programming using *if*, *then*, *else* and *do*, we won't have *go to*.

That was the beginning of what I thought was formal methods. That has slowly got into the system and now you don't see people writing really unstructured programs. The bigger problem I see now is not that everybody that comes out of university is unable to program, it's that they aren't able to communicate and write decent English. So if you can't write English, how can you write a specification that somebody else can understand and then implement.

John Fitzgerald: I believe we now call those "key transferable skills".

Margaret West: I think it was Richard Bornat who mentioned a test for students being able to program or not. When I was applying for jobs after graduating in mathematics, I was told "they'll probably make you program, because they think females are good at programming because it is like knitting."

Jawed Siddiqi: I think there's a very valid point about how education is moving towards training. It's not an issue of whether the young know better or the old know better, I think it is the pressure that we live in to try to make things that are supposedly more immediately exploitable. A system where students are customers means that at many times, it becomes much more difficult for universities to hang on.

When you talk about grammars, there is a whole series of courses that are now advertised as computing, rather than computer science. Given that it is (or was) one of the most popular courses, I think the range of material that's taught is huge and you would not expect that in any other discipline. There's a problem that some people are graduating who clearly have a computer science background underpinning their learning and there are those with supposedly immediately exploitable skills who aren't going to be much use in a few years time.

Cliff Jones: The line I use to "con" my students to come on my semantics course, is to talk about the half-life of knowledge. VB won't be there very long; but understanding the semantics of whatever language they are going to do, will be there for the rest of their career.

David Crocker: With the mass out-sourcing of programming jobs to low cost countries, I think there's a very good case to be made, that it's the higher skills of things like genuine design and real software engineering, rather than just programming, which are going to be the useful ones for students in the future.

3.6 Closing answer and comments

John Fitzgerald: I would like to finish with one question, which was sent to me in advance by Janos Korn, who is not a formal methods person, but asks a question that I think would be very appropriate to finish with.

Janos Korn: Thank you very much. I am an alien to formal methods, but I feel that it is probably very much relevant to the work I am interested in. In fact I value remarks and comments, that make me feel as though it is relevant — the verbosity of requirements; syntax and grammar and semantics, but I may be completely wrong. So if you allow me just for a few minutes to describe what I am interested in.

I am interested in the systemic view of the world, in other words, looking at the parts of the world, as consisting of interacting or interrelating objects or properties. This is a very general view that you can apply to anything. In fact, I believe that it is pervasive, empirical and indivisible. Hundreds of people at different universities are engaged in this, but the problem is that they are all extremely verbose and airy-fairy and talkative in terms of using abstract linguistics. What I am trying to do is to make this more concrete, relevant and structured and relatable to experience. In order to do this you need a symbolism to create models. The symbolism is not exclusively mathematics, but language. In fact, natural language matches the generality of this view, but natural language is full of problems, as we know. It is used as means of communication, but I am trying to use it as means of creating a model.

The problem is that the manifestations of the systemic view, with everything being an interrelated object, is so diverse with enormous variety, how can we adopt natural language? What

I believe we have to do is to use linguistic analysis to discover the smallest elements of natural language from which we can construct scenarios, which are formal and structured. These smallest elements are one and two phrase sentences, in other words, ordered pairs. I have developed this idea for some time, but my problem is that I lack the expertise in formal methods and of course development of software that can handle these complexities. This is where I think that formal methods perhaps could be a way towards the further development of this idea of formalized natural language.

John Fitzgerald: Thank you very much. The question that comes out of this is whether, over the last 25 years, formalists have done enough to relate the work that they have been doing in their laboratories to the much wider systems engineering process, rather then just the particular specialization on software? So I would like to ask you individually to comment on this issue.

Cliff Jones: No, we have not done enough!

Jean-Raymond Abrial: This is precisely the goal of Event B and also what precisely what I learned from Action Systems and from the people at the Finnish school. I think that studying the global system, not only the software part, but also the environment of it and having a model of this entire closed body is extremely important.

Bernie Cohen: It cannot be closed if there are people in it. That is the point.

Jean-Raymond Abrial: People are part of it.

Bernie Cohen: But it cannot be closed if there are people in it, it must be open.

John Fitzgerald: Any other comments? John, please.

John Tucker: I have a comment about theoretical models, the sort of classical corpus that Richard dislikes and classical corpus that I personally think is extremely important.

Over the course of many years, various notions have come out of computer science, or speculations on problems that are recognized as computer science. Grammars are examples, so are neural nets and of course many more things, not least a great number of process algebras.

Most of these things have arisen because we are all working on a certain kind of computing technology; they have arisen to try to produce understanding. But the first thing I would say is that, ignoring the responsibility of getting these tools integrated into the wider picture of design, I believe most of these tools will have a future becoming mainstream in the modeling of physical systems.

The physical systems that I am talking about may be indeed in pure science, but they could equally well be in engineering science. It is quite obvious that if you look at very theoretical work, for example on new models of computation, all the expertise and knowledge of formal description languages and their semantics, has a little role to play, to express certain intuitions about the way in which quantum mechanics and various other things can be described.

Although this is a highly technical area, it is nevertheless those tools that are gradually migrating, slowly but surely, into areas on the fringes of physics. Similarly, you can see these things in engineering. If you go to any big engineering department, you realize that the core business of virtually everybody in that engineering department (whether it is geology, aeronautics or chip design) is registered by the production of software models.

These software models typically come from continuous models, but as time goes by we see endless experiments in non-continuous modeling, for example of turbulence and various other things, using fancy discrete space models. I don't think we have seen the beginning of the future of our methods.

It is quite interesting that many technical things in the history of applied mathematics started as studies of practical problems in engineering, very concrete problems. For example, Chebyshev polynomial approximation — incredibly important. What was Chebyshev doing? He was trying to understand the gearing of railroads in Russia. So a problem to do with engineering of the different gears and the connecting rods between the wheels, needed certain kind of approximation and that is Chebyshev polynomial.

We see lots of these examples throughout time; I think it is just a matter of time before formal syntax, formal semantics, formal methods and particularly specification, start to play a fundamental role as they start to migrate out of our community into other scientific communities. Just as logic has done.

John Fitzgerald: I would like to thank all the members of the panel.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[ACJ72] C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972.

[Acz82] P. Aczel. A note on program verification. (private communication) Manuscript, Manchester, January 1982.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice Hall International, 1982.

[BJMN87] D. Bjørner, C. B. Jones, M. Mac an Airchinnigh, and E. J. Neuhold, editors. *VDM – A Formal Definition at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BW71] H. Bekič and K. Walk. Formalization of storage properties. In *[Eng71]*, pages 28–61. 1971.

[dBS69] J. W. de Bakker and D. Scott. A theory of programs. Manuscript notes for IBM Seminar, Vienna, August 1969.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods.* Cambridge University Press, 2001.

[DS90] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990. ISBN 0-387-96957-8, 3-540-96957-8.

[Eng71] E. Engeler. *Symposium on Semantics of Algorithmic Languages.* Number 188 in Lecture Notes in Mathematics. Springer-Verlag, 1971.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development.* Cambridge University Press, 1998.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems.* Springer, 2005.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC, SRC, July 1985.

[Gro92] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992. ISBN 0-13-752833-7.

[Gro95] The RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 1995. ISBN 0-13-752700-4.

[Gut75] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Computer Systems Research Group, September 1975. CSRG-59.

[Hay86] I. Hayes, editor. *Specification Case Studies.* Prentice-Hall International, 1986.

[HJN94] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *ACM Software Engineering News*, 19(3):75–81, July 1994.

[JL71] C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In *[Eng71]*, pages 178–211. 1971.

[Jon70] C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.

[Jon72a] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon72b] C. B. Jones. Operations and formal development. Technical Report TN 9004, IBM Laboratory, Hursley, September 1972.

[Jon73] C. B. Jones. Formal development of programs. Technical Report 12.117, IBM Laboratory Hursley, June 1973.

[Jon77] C. B. Jones. Implementation bias in constructive specification of abstract objects. typescript, September 1977.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, 1980. ISBN 0-13-821884-6.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986. ISBN 0-13-880717-5.

[Jon99] C. B. Jones. Scientific decisions which characterise VDM. In *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 28–47. Springer-Verlag, 1999.

[Jon01] C. B. Jones. The transition from VDL to VDM. *JUCS*, 7(8):631–640, 2001.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon06] Cliff B. Jones. Reasoning About Partial Functions in the Formal Development of Programs. *Proceedings of AVOCS'05*, volume 145 of Electronic Notes in Theoretical Computer Science, pages 3–25, 2006.

[Kin69] J. C. King. *A Program Verifier.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[Kin71] J. C. King. A program verifier. In C. V. Freiman, editor, *Information Processing 71*, pages 234–249. North-Holland, 1971. Proceedings of IFIP'71.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[Luc81] P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Research and Development*, 25(5):549–561, September 1981.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3.* Pergamon Press, 1969.

[Mid93] Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification.* Chapman and Hall, 1993.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[Oll75] A. Ollengren. *A Definition of Programming Languages by Interpreting Automata*. Academic Press, 1975.

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[Zil74] S. N. Zilles. Abstract specifications for data types. Technical Report 11, M.I.T. Progress Report, 1974.

John Fitzgerald, Cliff Jones, Ian Hayes, J-R Abrial, John Tucker

Jonathan Bowen (FACS Chair) John Fitzgerald (FME Chair)

Peter Mosses (background) during question time

Audience

Formal Methods Going Mainstream – Cost, Benefits and Experiences: Report on the ForTia Industry Day at FM05

Tiziana Margaria, Bernhard Schätz and Marcel Verhoef

The Formal Techniques Industrial Association ForTIA, a subgroup of Formal Methods Europe, organized an "Industry Day" as a parallel track to the 13th International Symposium on Formal Methods held at the University of Newcastle upon Tyne on 20 July 2005. The purpose of the Industry Day is to focus on the perceived pros and cons of formal methods by end-users, not necessarily the proposition(s) that were made by the technology providers. The aim is to understand how formal techniques can be introduced successfully in an industrial setting and how common pitfalls can be prevented, pitfalls that might not necessarily have anything to do with the formal technique itself.

The theme for I-Day 2005 was "*Formal Methods Going Mainstream - Costs, Benefits and Experiences*". Six invited speakers provided us with their insight into this theme; each was given half an hour to reflect on their experiences bringing formal techniques to the forefront of their business. SAP Research kindly sponsored the event. The Industry Day was closed with a panel (pictured above) discussion that also included the conference key-note speaker Matai Joseph (Tata Research Development and Design Centre), which resulted in a lively debate with the 75 people strong audience. Despite a substantial publicity campaign in the UK set up by the local organizers, attendance from industry (non-conference attendees) was low. Nevertheless, the returned evaluation forms indicated that the talks were of high quality. They indicated that Industry Day was a success and that the event should be continued in its present form. FM 2006 at McMaster (Canada), has already kindly agreed to host another I-Day; the theme for that event will be "Security & Trust".

John Harrison (pictured left) of Intel Corporation (USA) kicked off, giving an eye-opening story on the role of formal methods in the verification of floating point units. Intel wrote off 475 million USD to cover damages for the incident with the incorrect division in early Pentium processors (also known as the FDIV bug). A similar problem in current chip designs would be much more costly. Chip designs are getting more complex, but the associated testing problem is growing even faster in size and complexity. Traditional testing techniques are not sufficiently powerful and formal verification techniques can sometimes offer a solution. Several techniques are used in the hardware industry today such as ACL2, COQ and PVS. Since the FDIV bug, formal verification has become almost standard practice. Twenty percent of the Pentium IV design was formally verified and many high-quality bugs were discovered before "first silicon". The HOL Light theorem prover was used by John and his team to verify the floating point operations of the Itanium processor. As can be expected, several bugs were found in the design, but the verification – in combination with the high-level specification of the algorithm – also increased the problem understanding which eventually let to several improvements to the design.

Christian Scheidler (pictured left) of Daimler Chrysler Research and Technology (Germany) presented his experiences using state of the art model checkers in the automotive industry. The aim of the EASIS project in which he is involved is to determine whether model checkers can replace traditional testing in embedded system design. Model checkers are attractive from an industrial perspective because it is to a large extent push-button technology. The state space is explored exhaustively without the need to create test vectors manually and if an error is found, then the counter example is automatically produced. However, model checking does not provide a proof as a result when no errors are found. One has to trust the quality of the model checker itself. Furthermore, it is not known *a priori* how long it takes to search or how much memory is required to construct the state space. These problems seem to hinder industrial acceptance. A few realistic case studies were performed using the EmbeddedValidator tool suite, which provides an integrated platform that couples Matlab/Simulink, dSPACE and two proof engines: VIS and Prover. In total 67 out of 81 requirements could be automatically verified and residual errors where found in 7 of them. Despite some restrictions in the notation and limitations in the tools, the technology seemed to work quite well in practice and resource problems were not experienced during the experiments. This is partly due to the application of a collection of standard properties commonly expected to hold for the models under development, liberating the developers from formalizing these properties on their own. The amount of time needed to create the formal models was acceptable in an industrial setting and it was possible to achieve this without in-depth formal methods knowledge.

Guy Broadfoot (pictured left) of Verum Consultants (The Netherlands) reported on the use of their patented Analytic Software Design method. First, Sequence-based Specification (SBS) is used to create a set of inherently consistent and complete requirements. Noteworthy, these specifications are free from mathematical notation, which makes it easy to communicate with critical project stakeholders. The ASD Model Generator is used to derive CSP models and designs from the Sequence-based Specification, which in turn can be analyzed using the FDR model checker. The ASD Code Generator can generate significant amounts of program source code automatically from these verified designs. The ASD method was used to design and implement the control software of a magnetic levitation device which is a subsystem that is used for high accuracy positioning of objects in a production cell, e.g. to move a silicon wafer in a waferstepper. During the analysis and design phase, 400 defects where found by formal verification and removed before coding started. In total 18000 lines of C++ code were generated automatically, which represents 90% of the total code base. 35 man weeks of effort was spent on the complete project leading to an overall productivity of 12 lines of code per hour, which is three times higher than during conventional development in C++. After delivery only 5 defects where detected, mostly related to errors in the handwritten "glue" code, leading to a rate of only 0.28 defects per KLOC.

Gerrit Muller (pictured left) of the Embedded System Institute (The Netherlands) gave a very provocative yet inspiring talk on the relationship between formal methods and system engineering of complex computerized systems. He believes that the meaning of the word "formal" has become overloaded and very unclear, in particular in industry. The word is often used to qualify the development process, not necessarily the mathematical rigor that is the topic of the FM conference. He claims that formal methods are well suited to prescribed homogeneous domains, but that systems engineering, which integrates specialized engineering disciplines, is by nature informal rather than formal. He used the development of a wafer stepper as an example to illustrate this point. He zoomed in on a concrete property of the system, the required optical resolution, and he showed that many potential decisions in many design dimensions can affect this property by orders of magnitude. He believes that it is impossible to create a (formal) model that captures all these aspects in such a way that it can be usefully subjected to rigorous mathematical analysis. He conjectures that the added value of formal methods is primarily the skills of the people using them: they are analytical, structured, firm in principle and consistent. Using these skills, these individuals can play an important role in the informal multi-disciplinary system engineering process, but not necessarily using mathematical models or applying rigorous analysis techniques.

Alexander Pretschner (pictured left) of ETH Zürich (Switzerland) took a critical look at model-based testing (MBT). The proposition of MBT is that the quality of the test process can be increased by creating abstract formal models of the system under test. These models are then used to create test sets (semi-) automatically, which leads to better coverage and more errors found. Since this is a largely automated process, test efficiency is also increased. He has tried to find evidence to support the perceived benefits of this approach and he concludes that such evidence in fact does not exist. Virtually no quantitative comparative research to traditional testing and inspection approaches has been performed. He executed a few experiments in an industrial setting to fill this gap. The results were surprising; the rate of errors found in both the system requirements and the code indeed increased compared to traditional testing, but the automation did not improve the effectiveness as expected. The abstraction of the test model basically limits the potential coverage; automation (generation of test case) is limited to the syntactic level. Although the study is too small to make generic conclusions, it is already clear from the experiments that the benefits of MBT should not be taken for granted: While MBT does not in general outperform traditional means of testing, the conformance of implementations, the development of an explicit model of the system under test and the generation of associated test cases provide additional quality assurance for the system specification.

Margus Veanus (pictured left) of Microsoft (USA) presented a practical model based testing tool called SpecExplorer. This tool enables modelling and automatic testing of concurrent object-oriented systems written in Spec#. SpecExplorer is being used daily by several Microsoft product groups to test operating system components and Web service infrastructure. As an experiment, they had two teams test some components of the Windows operating system, one using a traditional approach, the other using SpecExplorer. The model-based approach supported by SpecExplorer found roughly ten times more errors than the traditional approach. The amount of effort spend in total was roughly identical for both, but model-based testing helped to discover two times more design issues than bugs in the implementation, so it paid of already in the design phase to use this technique. Only half of the errors found were actual errors in the system under test (SUT); the other half was due to mistakes in the informal requirements, the model itself or bugs in the test harness. However, the errors that were found in the system under test where deep system-level bugs for which manual test cases would have been hard to construct. The comparison also demonstrated to the developers and testers that code coverage is a poor measure for testing concurrent software; often a single execution thread gives the same coverage. In fact, Margus claims that a good measure for code coverage does not yet exist.

The panel discussion led to the conclusion that Formal Methods are still far from main stream technology at the moment. Instead, they seem to play an increasingly important role in certain niche areas. Their economic value is certainly demonstrated in those cases, although in general real cost / benefit analysis and comparisons to other approaches are seldom made or are at best very limited in scope. There is a responsibility for the Formal Methods community to make their impact more quantifiable; this requires gathering more quantitative evidence from experiments, but typically this is rejected as "interesting research". There seems to be a big difference in mental attitude in the hardware and software communities. In the former, the use of formal techniques is well established, possibly because product liability claims are of real economic significance. In the software community, product liability is typically waived and the end-users still seems to accept that fact. The open question raised in the discussion was whether or not the uptake of formal methods in main stream software engineering is indeed hindered by that fact. The suggestion was made that in particular in the area of embedded software, where the borderline between hardware and software is inherently less obvious, this attitude is in fact changing. The quality demands posed on those type of systems, for example in the automotive domain, are typically identical to hardware and product liability is indeed a real concern here. Some participants claimed that "correct by construction" is now feasible at a similar cost to traditional approaches but providing inherently better quality levels than traditional development. In the long run, the use of formal methods is not a matter of choice but a matter of survival in the rapidly evolving economic market place.

More information on ForTIA can be found at http://www.fortia.org. This web-site also contains slides of all the talks mentioned above. Extended abstracts of the talks can be found in the proceedings of FM 2005, volume 3582 of the well-known Lecture Notes in Computer Science series.

Joining Other Societies and Groups

London Mathematical Society
http://www.lms.ac.uk/contact/membership.html

Formal Methods Europe
http://www.fmeurope.org/fme/member.htm

European Association for Theoretical Computer Science
http://www.eatcs.org/organization/membership.html#how_to_join

Association for Computing Machinery
https://campus.acm.org/Public/QuickJoin/interim.cfm

IEEE Computer Society
www.computer.org/join/

The British Computer Society
www.bcs.org/bcs/join/

Model-Oriented Specifications

Tim Clement & Cliff B Jones

This was a *draft* of a chapter for a book on 'Algebraic Foundations of System Specification' being edited by Egidio Astesiano, Hans-Jörg Kreowski and Bernd Krieg-Brückner. (published as [AKKB99]). It has never been published. Here only Sections 1–3 are included.

Contents

1 Introduction

The aim of this chapter is to compare the algebraic approaches to system specification with the model-based approach typified by the specification languages VDM and Z and the Abstract Machine Notation of Abrial. This is done by considering a series of examples, each illustrating a particular concern that arises in specification. The first of these, bags, is a simple example that illustrates the different approaches without involving any particular technical difficulty, while the traversable stack raises issues of operations which are partial, and the third example in Section 5 introduces operations which can be nondeterministic. In Section 6, some lessons are drawn from these different specifications.

For concreteness, the examples of model-based specifications are presented using VDM. However, they use only a subset of the language which should be accessible to most readers (a summary of notation is provided in Appendix A); all examples could equally well –with little change beyond alterations in syntax– be presented in one of the other specification languages mentioned above. (Editor's note: Sections 5 and 6 and Appendix A do not appear in the newsletter.)

1.1 Abstract types

The usual view of an abstract type in a programming language is of a definition which names a type (so that the rest of the program can declare identifiers of that type) and provides a set of operations on it, but hides the way that the values of the type and the operations on them are implemented. All the user of the type can rely on is some properties of the operations, expressed as properties of results of types other than that of the abstract type being defined (the **visible behaviour**, since this is all that the user can see: the previously defined types are **visible** while the abstract type itself is said to be **hidden**). For example, a stack type definition says that pushing a series of values onto a stack and then repeatedly popping it returns the values pushed in reverse order.

On the other hand, the informal understanding of an abstract type is of a collection of values (so for example a stack is a pile of its elements, which are drawn from some arbitrary set) which are transformed by the operations defined on it (so pushing a value onto a stack adds it to the top of the pile, and popping removes the pile top). The visible behaviour is a consequence of the effects of the operations on the values. Thus, intuitively, a $push$ followed by a pop returns the value pushed and the stack it was pushed on. As a result, a further pop returns the value previously pushed and so on, as observed. The rôle of the specification is to capture this informal understanding in a formal definition. The visible behaviour can then be derived, and implementations justified as providing the required visible behaviour.

1.2 The approaches to specification

The model based approach to defining abstract types uses a rich selection of predefined types and ways of constructing new types from them to define a type that is a formal model of the informal collection of values. This means that for each informal value there is a single formal value that represents it, and conversely, each formal value has a unique informal counterpart: that is, there is an informal bijection between the informal and formal views. It is usually helpful to describe this bijection explicitly. Informal ideas of operations are then formalized as functions mapping the formal counterpart of the input to the formal counterpart of the result. Although there is always a choice of models (since formally anything of the right cardinality will do), it helps to select one which reflects any structure seen in the informal values. In this way, the effects of the operations on the informal structure can be mirrored in their effects on the formal structure. Properties of the visible behaviour must be derived from the operation definitions.

In the property based approach, a name is introduced for the abstract type, which is then defined up to isomorphism by giving properties of operations on the type. (As a consequence, everything true of the model will be deducible from the axioms.) A suitable set of axioms can be achieved by writing down properties of the intended model, introducing operations expected for the type as necessary to express them, and trying to think of non-isomorphic models where the properties would hold, adding axioms until no unwanted models are left. (The range of possible models will have an effect on this of course. Normally types will be interpreted as sets and operations as functions as in the model based approach.) In most cases, the set of values of the type can be characterized using only a few of its operations. These will include enough operations with the type as their result type (**constructors**) to create all the values. Further operations can be added once the basic definition of the type has been given. The operations will usually have parameters or results of types other than that being defined; these are assumed to have been defined previously in the same way. The properties of the visible values that the user can rely on can then be deduced as consequences of the axioms.

Like the property based approach, the algebraic approach uses axioms to formalize the informal model. The difference is that the model specified by a set of axioms is the initial model of the category of models that it possesses, rather than the unique model that it characterizes. (Forming a category needs a suitable notion of homomorphism between models.) The initial model necessarily exists for an arbitrary set of axioms only if the axiom language is quite restricted: negation is forbidden, for example. The usual applications of algebraic specification are based on languages with equality, and the initial model can be constructed by taking the sets of terms which are provably equal as values. The art of defining an abstract type algebraically is to find sufficient constructors to produce all the terms that are needed, and sufficient equations on them to make this term model isomorphic to the intended one. Equality properties (which are the most common) can of course be declared from the axioms of the specification. Other properties can be deduced from the equations and the initiality, which amongst other things justifies an induction schema.

2 Bags

Bags, also known as multisets, provide a simple example of an abstract type that can be used to illustrate the three approaches to abstract type definition. The informal view of a bag is of an unordered collection of elements where each element can appear many times: the name evokes an image of a container holding an assortment of elements. The bag containing no elements is said to be empty. Operations that might be applied to bags include adding and removing single elements (which increase or decrease the number of occurrences by one, assuming in the case of removal that the element occurs in the bag), determining the number of occurrences of an element, or taking the union of two bags.

2.1 The model based approach

The important aspect of the informal view as far as modelling bags is concerned is that each distinct element in the bag can be associated with its number of occurrences. This structure can be formalized as a map from the elements (an arbitrary set) to numbers which are always strictly positive.

$$Bag = X \xrightarrow{m} \mathbb{N}_1$$

By considering examples, it is easy to see that each informal bag has a unique formal counterpart and *vice versa*.

Each operation takes values of the type and returns new values: the formal definition should capture this by mapping the formal value corresponding to the input to the one corresponding to the result. Because the formal model reflects the intuitive view of the structure of bags, the intuitive effect of the operations on that structure can be reflected in their formal definitions

$$empty \colon Bag$$

$$empty = \{\,\}$$

$$add : X \times Bag \to Bag$$

$$add(x, b) \quad \triangleq \quad \textsf{if } x \in \textsf{dom}\, b \textsf{ then } b \dagger \{x \mapsto b(x) + 1\} \textsf{ else } b \cup \{x \mapsto 1\}$$

$$remove : X \times Bag \rightarrow Bag$$

$$\begin{aligned} remove(x, b) \;\; \triangleq \;\; & \textsf{if } x \in \textsf{dom}\, b \\ & \textsf{then if } b(x) > 1 \;\; \textsf{then } b \dagger \{x \mapsto b(x) - 1\} \;\; \textsf{else } \{x\} \mathbin{⩤} b \\ & \textsf{else } b \end{aligned}$$

$$count : X \times Bag \rightarrow \mathbb{N}$$

$$count(x, b) \;\; \triangleq \;\; \textsf{if } x \in \textsf{dom}\, b \;\; \textsf{then } b(x) \;\; \textsf{else } 0$$

$$union : Bag \times Bag \rightarrow Bag$$

$$union(b_1, b_2) \;\; \triangleq \;\; \{x \mapsto count(x, b_1) + count(x, b_2) \mid x \in \textsf{dom}\, b_1 \cup \textsf{dom}\, b_2\}$$

For example, that add and $remove$ increment and decrement the number of occurrences of an element is clearly apparent.

An alternative, but less natural, model for bags associates every possible element with a number of occurrences; those elements not in the bag being given a count of zero. This is clearly isomorphic to the previous model, and is formalized as

$$Bag = X \xrightarrow{m} \mathbb{N}$$

$$\textsf{inv-}Bag(b) \;\; \triangleq \;\; \textsf{dom}\, b = X$$

The invariant insists that every element has an associated number.[1] The benefits of totality are seen in the operation definitions, where checks for being in the domain are no longer needed. (Previously, the definition might have been undefined in some cases without them.)

$$empty\colon Bag$$

$$empty = \{x \mapsto 0 \mid x\colon X\}$$

$$add : X \times Bag \rightarrow Bag$$

$$add(x, b) \;\; \triangleq \;\; b \dagger \{x \mapsto b(x) + 1\}$$

$$remove : X \times Bag \rightarrow Bag$$

$$remove(x, b) \;\; \triangleq \;\; \textsf{if } b(x) > 0 \;\; \textsf{then } b \dagger \{x \mapsto b(x) - 1\} \;\; \textsf{else } b$$

$$count(x, b) \;\; \triangleq \;\; b(x)$$

[1]This is not strictly legal in VDM if X is infinite.

The properties that might prove useful to a user of the type define how many times an element occurs after any series of operations: they include

$$\begin{aligned}&count(x, empty) = 0\\&count(x, add(x, b)) = count(x, b) + 1\\&count(x, union(b_1, b_2)) = count(x, b_1) + count(x, b_2)\end{aligned}$$

These can all be proved simply from the definitions. This can be extended to the total number of elements in the bag.

2.2 The property based approach

Consider how a model of bags might be characterized by axioms. Since the purpose of a bag is to maintain the number of occurrences of its elements, if the counts of all elements in two bags are the same, then the bags are the same. This is formalized by introducing the operation $count$ and writing

$$(\forall x \cdot count(x, b_1) = count(x, b_2)) \;\Rightarrow\; b_1 = b_2$$

As well as the intended one, models with this property include some with only one element of the abstract type. The unwanted models can be eliminated by introducing an add operation and ensuring that it adds to the number of occurrences of its argument as expected. This can be axiomatized by

$$\begin{aligned}&count(x, add(x, b)) = count(x, b) + 1\\&x_1 \neq x_2 \;\Rightarrow\; count(x_1, add(x_2, b)) = count(x_1, b)\end{aligned}$$

The only models of these axioms with any values of the type have all the values expected. There is still a model with no values, which can be ruled out by introducing the empty bag as a constant $empty$. The only remaining unintended models are those containing **junk**: values that cannot be produced by terms of the form $add^n(x_1, \ldots x_n, empty)$ (where x_i is the argument of the ith add in the term). These can be eliminated by an induction axiom schema

$$\frac{P[empty/b] \quad \forall x{:}\, X, b{:}\, Bag \cdot P[b] \;\Rightarrow\; P[add(x, b)]}{\forall b{:}\, Bag \cdot P[b]}$$

This last axiom is unlike the others: it is much more elaborate to state, involves the language of sets rather directly, and does not directly express the idea that there is no junk. Larch provides the useful shorthand

generated by $empty, add$

for this axiom, and it is adopted here.

Now the set of values of the type is defined up to isomorphism, but the definition of the $count$ operation is incomplete: the number of each element in the empty bag is not necessarily zero. This omission can be repaired by

$$count(x, empty) = 0$$

Now the set of values has been defined, attention can turn to the remaining operations. These can be defined by their effects on the element counts in the same way as add.

$$\begin{aligned}&count(x, remove(x, b)) = max(count(x, b) - 1, 0)\\&x_1 \neq x_2 \;\Rightarrow\; count(x_1, remove(x_2, b)) = count(x_1, b)\end{aligned}$$

$$count(x, union(b_1, b_2)) = count(x, b_1) + count(x, b_2)$$

The new axioms are added to those of the previous definition, so all previous theorems remain theorems. Because the previous axioms described the model up to isomorphism, the new ones should not change it.

As in model oriented specifications, there are other ways in which the definition could be constructed. For example, there is an induction principle for bags based on the operations $empty$, $union$ and $unit$, where $unit(x)$ gives the bag containing nothing but a single x. This can be expressed by

generated by $empty, unit, union$

and $unit$ can be defined by

$$count(x, unit(x)) = 1$$
$$x_1 \neq x_2 \;\Rightarrow\; count(x_1, unit(x_2)) = 0$$

The remaining operations are unchanged.

The useful properties that were derived in the model based approach are axioms in the definition, although other properties such as the total number of elements in the bag would have to be derived.

2.3 The algebraic approach

Bags can also be defined as the initial model of a set of equations. (This is the simplest language for which to establish the existence of an initial model.) As the property based definition implies, all bags can be constructed by applying add sufficiently often to $empty$. In doing this, the order in which the elements are added does not affect the final result, so any permutation of a series of adds is equal to any other. That is, the intended model of bags is isomorphic to a term model where all permutations of the values added in the term are in the same equivalence class. Because any permutation can be turned into any other by repeatedly swapping adjacent values, an initial model of the signature with the two operations

$$empty : \rightarrow Bag$$
$$add : X \times Bag \rightarrow Bag$$

and the single equation

$$add(x_1, add(x_2, b)) = add(x_2, add(x_1, b))$$

gives exactly these equivalence classes, and hence the intended model is also an initial model of this definition. It is thus an algebraic specification of bags with these two operations.

Further operations can be specified by recursion on the terms created by add and $empty$. For example, the operation $count$ should be defined if the abstract type is to be of much use. To do this equationally needs some supporting definitions. If a Boolean operation $==$ can be defined equationally on X, and the if _ then _ else _ operation is defined generically in $\mathbb{B}$ by

if $true$ then x_1 else $x_2 = x_1$
if $false$ then x_1 else $x_2 = x_2$

then $count$ can be defined by

$$count(x, empty) = 0$$
$$count(x_1, add(x_2, b)) =$$
$$\quad \text{if } x_1 == x_2 \text{ then } count(x, b) + 1 \text{ else } count(x, b)$$

Similarly, the remove operation can be defined by

$$\begin{aligned}
&\mathit{remove}(x, \mathit{empty}) = \mathit{empty}\\
&\mathit{remove}(x_1, \mathit{add}(x_2, b)) =\\
&\qquad \text{if } x_1 == x_2 \text{ then } b \text{ else } \mathit{add}(x_2, \mathit{remove}(x_1, b))
\end{aligned}$$

and union by

$$\begin{aligned}
&\mathit{union}(\mathit{empty}, b) = b\\
&\mathit{union}(\mathit{add}(x, b_1), b_2) = \mathit{add}(x, \mathit{union}(b_1, b_2))
\end{aligned}$$

An alternative is to use unit and union as the constructors. Every use of unit gives a different bag, but the order of building larger bags through union does not matter. The appropriate initial model for this signature is now defined by the equations

$$\begin{aligned}
&\mathit{union}(\mathit{empty}, b) = b\\
&\mathit{union}(b_1, b_2) = \mathit{union}(b_2, b_1)\\
&\mathit{union}(b_1, \mathit{union}(b_2, b_3)) = \mathit{union}(\mathit{union}(b_1, b_2), b_3)
\end{aligned}$$

The other operations can be defined by

$$\begin{aligned}
&\mathit{count}(x, \mathit{empty}) = 0\\
&\mathit{count}(x_1, \mathit{unit}(x_2)) = \text{if } x_1 == x_2 \text{ then } 1 \text{ else } 0\\
&\mathit{count}(x, \mathit{union}(b_1, b_2)) = \mathit{count}(b, x_1) + \mathit{count}(b, x_2)\\
\\
&\mathit{remove}(x, \mathit{empty}) = \mathit{empty}\\
&\mathit{remove}(x_1, \mathit{unit}(x_2)) = \text{if } x_1 == x_2 \text{ then } \mathit{empty} \text{ else } \mathit{unit}(x_2)\\
&\mathit{remove}(x, \mathit{union}(b_1, b_2)) =\\
&\qquad \text{if } \mathit{count}(x, b_1) == 0 \text{ then } \mathit{union}(b_1, \mathit{remove}(x, b_2)) \text{ else } \mathit{union}(\mathit{remove}(x, b_1), b_2)\\
\\
&\mathit{add}(x, b) = \mathit{union}(\mathit{unit}(x), b)
\end{aligned}$$

The choice of starting point makes a significant difference to the ease of the obvious axiomatization.

Of the properties established of the previous definitions,

$$\mathit{count}(x, \mathit{empty}) = 0$$

and

$$\mathit{count}(x, \mathit{add}(x, b)) = \mathit{count}(x, b) + 1$$

follows readily from the definitions. The property

$$\mathit{count}(x, \mathit{union}(b_1, b_2)) = \mathit{count}(x, b_1) + \mathit{count}(x, b_2)$$

must be established by induction. Initiality implies an induction principle involving all the constructors of the signature: in this case

$$\frac{P[\mathit{empty}/b] \qquad P[\mathit{unit}(x)/b] \ldots}{P[b]}$$

The expected induction principle, which was axiomatic in the property based approach, must be derived from this and the definitions of unit, remove and union.

3 Traversable stacks

The example in this section affords comparison of various ways of handling partial operations and pinpoints a limitation of the algebraic approach.

A traversable stack is like an ordinary stack (and hence provides $push$ and pop operations) but allows elements within the stack to be read by returning the value at a position defined by a cursor. Operations to move the cursor might include one to move it down the stack one step, and one to move it back to the top. It starts at the top of the stack, and can be moved just off the bottom of the stack but no further, and cannot be moved above the top of the stack. The $push$ and pop operations move the cursor position one element up or down the stack (so if only $push$es and pops are done, the cursor stays at the top of the traversable stack, which thus behaves like an ordinary stack). As an abstract type, traversable stacks are more complicated than bags. The principal issue to be addressed in the formalization is that the effect of a $read$ or a pop or a $down$ when the cursor is off the bottom of the stack is not obvious: the most natural solution may be to say that it is not allowed.

3.1 The model based approach

Since stacks maintain their elements in order, the obvious model is a sequence of values; the cursor can be represented by a number. It turns out to be more convenient when defining the operations to make the top of the stack the head of the sequence. The cursor position is best maintained as the distance from the top (as then $push$ and pop can leave it unchanged), and because VDM indexes sequences starting at 1, its range should be from 1 to one more than the stack length.

$$\begin{array}{lll} TStack :: & s & : X^* \\ & c & : \mathbb{N}_1 \end{array}$$

$$\mathsf{inv}\text{-}TStack(s, c) \quad \triangleq \quad c \leq \mathsf{len}\, s + 1$$

The formal definition of the effect of the operations on the components of the stack follows the informal one quite closely.

$$empty = \mathsf{mk}\text{-}Stack([\,], 1)$$

$$push : X \times TStack \to TStack$$

$$push(x, \mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(\mathsf{cons}(x, s), c)$$

$$pop : TStack \to TStack$$

$$pop(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(\mathsf{tl}\, s, c)$$

$$down : TStack \to TStack$$

$$down(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(s, c + 1)$$

$$to\text{-}top : TStack \to TStack$$

$$to\text{-}top(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(s, 1)$$

$$read : TStack \to X$$

$$read(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad s(c)$$

These definitions repay some further investigation. For example, if informally the cursor is off the stack, then formally the sequence is indexed beyond its limits, and the result of this, and hence the result of the operation, is undefined. Similarly, but more subtly, pop and $down$, if applied when the cursor is off the stack, attempt to construct a new stack with the cursor even further off. This violates the invariant, which is implicitly an assumption on the input and a requirement on the result of a function, and hence again the result is undefined. This corresponds exactly to the situations where informally the operations were not allowed, so this aspect of the intuition has also been captured. VDM and Z treat the verification of this rather differently. In Z, the domain of definition should be derived and compared with the intuition, as above, while in VDM the expected domain is formalized as a **precondition** on the operation, giving the definitions

$$pop : TStack \to TStack$$

$$pop(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(\mathsf{tl}\, s, c)$$

$$\mathsf{pre}\ c \leq \mathsf{len}\, s$$

$$down : TStack \to TStack$$

$$down(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad \mathsf{mk}\text{-}Stack(s, c + 1)$$

$$\mathsf{pre}\ c \leq \mathsf{len}\, s$$

$$read : TStack \to X$$

$$read(\mathsf{mk}\text{-}Stack(s, c)) \quad \triangleq \quad s(c)$$

$$\mathsf{pre}\ c \leq \mathsf{len}\, s$$

for these partial operations. The definedness of the result should then be proved under the assumption of the precondition. Formalizing “not allowed” as undefined works well when the partial operations are used in larger definitions. If they can be used where they are undefined, the result of the larger operation will be undefined, just as undefinedness propagated from the basic operations to those of traversable stacks. It remains not allowed to do anything not allowed, with no special treatment required in the larger definitions.

Informally, properties involving partial functions are only expected to hold when the operations are allowed. For example, a $push$ followed by a $down$ has the same effect as a $down$ followed by a $push$ but only if the $down$ is allowed. Formally, the precondition must be made explicit

$$\mathsf{pre}\text{-}down(s) \ \Rightarrow \ push(x, down(s)) = down(push(x, s))$$

However, since $down$ should only be used when the precondition holds, the condition should be checked anyway and so the property can be used with no extra work.

3.2 The property based approach

The properties of the traversable stack are less obvious than those of bags. To discover some, it is helpful to sketch the informal model, showing how each value can be produced by one or more

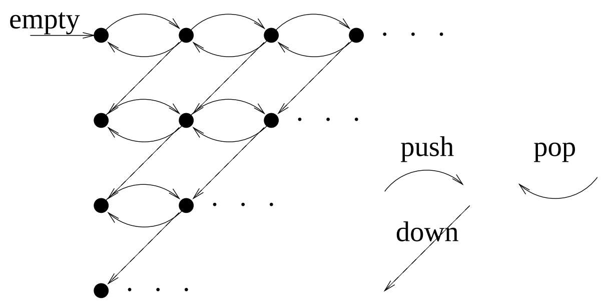


Figure 1: An informal view of the traversable stack

constructors from other values. (The operations $to\text{-}top$ and $read$ are omitted.) If the differences induced by pushing different values onto the stack are ignored, an informal view of the type is shown in figure 1.
The structure of the values themselves is not important in a property based definition. The difficulty of partiality in the operations is recorded here as the absence of $down$ and pop arrows from the values on the left hand side of the diagram.

This causes some problems in the formalization, because the usual assumption in property based and algebraic specification is that the models of operations are total functions. If this view is to be maintained when defining traversable stacks, then all operations must be made total, and the obvious way to do this is to add an explicit $error$ value to the type that will be the target of the missing arrows. However, this leads to a new value of the type which must have the operations defined on it (although informally an error might be expected to terminate a program). The obvious result is $error$ again: doing something not allowed will formally give an error.

The resulting informal picture can now be captured as axioms. A good first step is to identify a sufficient set of constructors for the type: that is, a set of constants and functions that informally will allow every value to be produced. This can be formalized as a generated by clause: in this case,

$$\text{generated by } empty, push, down$$

Axioms can then be used to equate the results produced by different routes to the same value. One simple axiom defines the effect of $down$ on $empty$

$$down(empty) = error$$

There are still models of this with a single value of the type. Guaranteeing at least two is easy

$$empty \neq error$$

More can be ensured by using the idea that pushing things onto a (traversable) stack gives a series of new stacks in the same way that adding one to a number always gives a new number.

$$s_1 \neq error \wedge push(x_1, s_1) = push(x_2, s_2) \;\Rightarrow\; x_1 = x_2 \wedge s_1 = s_2$$
$$push(x, s) \neq empty$$

These axioms guarantee the infinite chain of traversable stacks starting from $empty$ and generated by $push$ (the values on the top row of the diagram, which are essentially the ordinary stacks) but say nothing about the remaining chains, one for each value of c in the formal model of the proper subset, which are only reachable by $down$. One obvious property suggested by the diagram is

$$down(push(x_1, push(x_2, s))) = push(x_1, down(push(x_2, s)))$$

This remains true even when $s = \mathit{error}$. It is not sufficient to prevent down from producing error everywhere, but this can be ruled out by

$$s \neq \mathit{error} \;\Rightarrow\; \mathit{down}(\mathit{push}(x, s)) \neq \mathit{error}$$

(which subsumes $\mathit{push}(x, s) \neq \mathit{empty}$ above). This still leaves unfixed the effect of down on stacks where the cursor is at the bottom. This gap can be filled completely by

$$\mathit{down}(s) = \mathit{error} \;\Rightarrow\; \mathit{down}(\mathit{down}(\mathit{push}(x, s))) = \mathit{error}$$

Repeated application of this rule starting with $s = \mathit{empty}$ fills in all the down arrows on the left of the diagram. All that remains to define a unique model is to define the effect of the operations on error.

$$\begin{array}{l} \mathit{push}(x, \mathit{error}) = \mathit{error} \\ \mathit{down}(\mathit{error}) = \mathit{error} \end{array}$$

Now that the values and their constructors are characterized, attention can turn to the remaining operations. The $\mathit{to\text{-}top}$ operation can be defined by a simple recursion on the constructors

$$\begin{array}{l} \mathit{to\text{-}top}(\mathit{empty}) = \mathit{empty} \\ \mathit{to\text{-}top}(\mathit{error}) = \mathit{error} \\ \mathit{to\text{-}top}(\mathit{push}(x, s)) = \mathit{push}(x, \mathit{to\text{-}top}(s)) \\ \mathit{down}(s) \neq \mathit{error} \;\Rightarrow\; \mathit{to\text{-}top}(\mathit{down}(s)) = \mathit{to\text{-}top}(s) \end{array}$$

and the pop operation is defined similarly

$$\begin{array}{l} \mathit{pop}(\mathit{empty}) = \mathit{error} \\ \mathit{pop}(\mathit{push}(x, s)) = s \\ \mathit{pop}(\mathit{down}(s)) = \mathit{down}(\mathit{pop}(s)) \end{array}$$

The axiomatization of read is more difficult because this recursive pattern does not lead to a definition. The result of reading a stack produced by a push operation is not obvious because the cursor position depends on the prior operations. However, it is obvious that

$$\mathit{read}(\mathit{to\text{-}top}(\mathit{push}(x, s))) = x$$

Since down and pop both have the effect (in their different ways) of moving the cursor down when possible, it is also the case that

$$\mathit{read}(\mathit{down}(s))) = \mathit{read}(\mathit{pop}(s))$$

When the cursor is at the bottom of the stack, or the "stack" is error, then reading is also an error: that is, read is partial. To formalize this in the same way, a value error_X must be assumed in X, and an axiom can then be given

$$\mathit{down}(s) = \mathit{error} \;\Rightarrow\; \mathit{read}(s) = \mathit{error}_X$$

This is enough to define read in all cases. However, the error value in X was not foreseen when writing the earlier axioms. Formally, pushing error_X onto a stack should probably result in error, although again it seems like an artifact of the formalization that the question even arises. The earlier axioms should thus be modified to include

$$s_1 \neq \mathit{error} \wedge x_1 \neq \mathit{error}_X \wedge \mathit{push}(x_1, s_1) = \mathit{push}(x_2, s_2) \;\Rightarrow \\ \hfill x_1 = x_2 \wedge s_1 = s_2$$

$$\mathit{push}(\mathit{error}_X, s) = \mathit{error}$$

This approach of totalizing the operations on the abstract type is clearly not entirely satisfactory. It introduces a gap between the intuitive view of the type, which contains only “proper” values, and its formalization which adds the error values, and as $read$ shows, the need for extra values tends to propagate into all types. Once part of the type, the extra value or values must be treated by the axioms, which tends to make them less concise because errors must be treated as special cases. Also, because the error values have no intuitive counterpart their effect on the axioms is easy to overlook (as it was in early drafts of this paper!).

The obvious solution is to allow models with partial functions, but it must be decided when a model satisfies an axiom now some of its terms may be undefined. This means looking at the semantics of the language of axioms, and there are several possible treatments. One convenient choice is to take a strong view of equality: two terms are equal if they are both defined and have the same value, or if they are both undefined. To define the model uniquely, it is also necessary to be able to specify where a function is defined. This can be done by defining a predicate $D(t)$ to hold if t is to be defined.

Using this language, the desired model of traversable stacks and their constructors can be defined by

generated by $empty, push, down$

$$\begin{array}{l}
\neg D(down(empty)) \\
D(empty) \\
push(x_1, s_1) = push(x_2, s_2) \;\Rightarrow\; x_1 = x_2 \wedge s_1 = s_2 \\
down(push(x_1, push(x_2, s))) = push(x_1, down(push(x_2, s))) \\
D(down(push(x, s))) \\
D(down(down(push(x, s)))) \;\Rightarrow\; D(down(s))
\end{array}$$

The relationship between these axioms and those which defined the total model is straightforward: axioms which involved $error$ become axioms for D (and it is convenient to express the contrapositive of the last of these), while the other axioms are essentially unchanged. (They are presented in the same order for ease of comparison.) There is some simplification because the type no longer includes an error value (and $read$ will no longer return one), so this does not have to be eliminated explicitly from consideration.

3.3 The algebraic approach

To obtain an algebraic definition of the traversable stack, it again helps to use the informal model of the type and its constructors. As with the property based approach, the problem of the informal partiality of the operations must be resolved, and because in the usual initial approach operations are modelled as total functions, this involves the introduction of error values.

As usual, the strategy for the definition is to identify constructors and give enough equations to select the appropriate term model. The constructors used in the property based definition can be tried as the basis of an algebraic definition. The only obvious general equality is that $down$ and $push$ commute on nearly all values.

$$down(push(x_1, push(x_2, s))) = push(x_1, down(push(x_2, s)))$$

This might be considered enough to define a satisfactory initial model. The term model provides –for each proper stack– the equivalence class of ways of building it (all interleavings of the $push$es and $down$s where the number of $down$s in any prefix does not exceed the number of $push$es). On top of this, there are many singleton equivalence classes: $\{down(empty)\}$, $\{down(down(push(empty)))\}$.

Each of these represents an error: unlike the property based specification there are many different errors. Having multiple error values makes little difference because none of them are based on intuition in any case. This approach does not work in the property based specification because some way of referring to the error values is needed to write the axioms rendered unnecessary by initiality here, and identifying them all with a single value is the easiest way. If an initial specification with a single error value is preferred, further axioms can be added

$$\begin{aligned}&\mathit{down}(\mathit{empty}) = \mathit{error}\\ &\mathit{down}(s) = \mathit{error} \;\Rightarrow\; \mathit{down}(\mathit{down}(\mathit{push}(x,s))) = \mathit{error}\end{aligned}$$

The last of these is a conditional equation. The category of algebras which model a set of conditional equations also has an initial object so this is permissible. (The initial term model just satisfies all equations deducible from the axioms, giving in this case an appropriate model for traversable stacks.) Conditional equations are known to be more powerful than equations, in the sense that a given algebra may be characterizable as the initial model of a finite set of conditional equations, but not as the initial model of a set of equations on the same operations [TWW82, Ore79]. The algebra of traversable stacks with operations empty, error, push and down with a single error value seems to be of that kind: the theory must contain equations of the form $\mathit{down}^{n+1}(\mathit{push}^n(\mathit{empty})) = \mathit{error}$ for all n, and there seems to be no way to obtain this from finitely many equations on these operations. (A proof that this is so for a simplified traversable stack is given in [TWW82]: [Kap79] shows how the traversable stack can be axiomatized equationally by using extra operations.)

The remaining operations can be defined much as before to complete the specification.

$$\begin{aligned}&\mathit{pop}(\mathit{empty}) = \mathit{error}\\ &\mathit{pop}(\mathit{error}) = \mathit{error}\\ &\mathit{pop}(\mathit{push}(x,s)) = s\\ &\mathit{pop}(\mathit{down}(s)) = \mathit{down}(\mathit{pop}(s))\\[1em] &\mathit{to\text{-}top}(\mathit{empty}) = \mathit{empty}\\ &\mathit{to\text{-}top}(\mathit{error}) = \mathit{error}\\ &\mathit{to\text{-}top}(\mathit{push}(x,s)) = \mathit{push}(x,\mathit{to\text{-}top}(s))\\ &\mathit{down}(s) \neq \mathit{error} \;\Rightarrow\; \mathit{to\text{-}top}(\mathit{down}(s)) = \mathit{to\text{-}top}(s)\\[1em] &\mathit{read}(\mathit{to\text{-}top}(\mathit{push}(x,s))) = x\\ &\mathit{read}(\mathit{down}(s))) = \mathit{read}(\mathit{pop}(s))\end{aligned}$$

There are other ways of dealing with partial operations. Categories of models where the functions are partial can be defined by introducing homomorphisms which preserve the values of defined terms in the usual way, and also preserve the definedness of terms. If a signature is fixed and the axioms are all of the form

$$\bigwedge_i D(t_i) \wedge \bigwedge_j (D(u_j) \wedge u_j = v_j) \;\Rightarrow\; C$$

where C is of the form $D(t)$ or $t = t'$, then the category has an initial model [BW82], so it is possible to take the algebraic approach to the specification of partial operations. As usual, the initial model contains no junk and has no confusion. Functions are defined only if they must be to satisfy the D axioms. The effect is that initiality plus the positive axioms of the property based approach

$$D(empty)$$
$$D(down(push(x, s)))$$
$$D(down(down(push(x, s)))) \;\Rightarrow\; D(down(s))$$

$$down(push(x_1, push(x_2, s))) = push(x_1, down(push(x_2, s)))$$

gives the same model as the property based definition.

An alternative approach is to retain the idea that operations are modelled as total functions, but to ameliorate the unpleasantness of the error values. Order sorted algebras [GM92] do this by allowing one type to be defined to be a subset of another type. Values of a subtype can be treated as values of a supertype. In the initial model, the sets are as small as possible consistent with the axioms and the idea that a supertype contains all the values that the subtype does. The obvious way to use this is to define the type of traversable stacks to be a subset of the type of traversable stacks plus errors (written $TStack \leq ErrTStack$). There is thus both a type that corresponds to the intuitive view and a type that contains the results of the "partial" operations. The types of the operations reflect this

$$empty\colon \to TStack$$
$$error\colon \to ErrTStackpush\colon X \times TStack \to TStack$$
$$down\colon TStack \to ErrTStack$$

The axioms can be simplified by quantifying only over the proper values where appropriate. For example, if traversable stacks are defined by the single axiom

$$\forall s\colon TStack \cdot down(push(x_1, push(x_2, s))) = push(x_1, down(push(x_2, s)))$$

(where the range of quantification needs to be made explicit now there is more than one kind of stack) then in the term model $ErrTStack$ is essentially the same set that this equation defined for $TStack$ in the first algebraic definition, while $TStack$ is now the proper values.

There is one technical issue: the equation given above does not seem to be type correct: $down$ returns an $ErrTStack$ whereas $push$ expects a $TStack$. Implicitly, though, an order sorted algebra contains **retracts** r from each supersort to each of its subsorts, defined in this case by

$$r\colon ErrTStack \to TStack$$
$$\forall s\colon TStack \cdot r(s) = s$$

(Note that the equation applies only to values of the subtype in the supertype.) It inserts retracts where necessary, so the equation is really

$$\forall s\colon TStack \cdot down(push(x_1, push(x_2, s))) = push(x_1, r(down(push(x_2, s))))$$

Provided the result of $down$ is a proper value, the retract definition can be used to eliminate it from the term. If it is not, then the use of the retract cannot be removed from the term $r(down(push(x_2, s)))$, so equivalence classes containing only terms with uses of retracts represent error values.

As an alternative, it may be possible to identify a subtype of the informal type which is the domain of the partial operation, and then define the operation as a total function over that type. For example, $down$ is defined on those traversable stacks where the cursor is not at the bottom. This can be captured by the signature

$$NETStack \leq TStack$$

$$empty\colon \to TStack$$
$$push\colon X \times TStack \to NETStack$$
$$down\colon NETStack \to TStack$$

Since there are no partial operations to totalize, there is no need for error values. The intended model is again the initial model of the single axiom

$$\forall x\colon X\,,\, s\colon NETStack \cdot push(x, down(s)) = down(push(x, s))$$

where $TStack$ has the traversable stack values and $NETStack$ those values where the cursor is not at the bottom. The definition of initiality is vital in getting the right model of stacks: in other models, $empty$ can be an $NETStack$ value, and the resulting set of values is rather different. Order sorted algebras could be used as a way of giving property based definitions, but not without some explicit way of saying that some values are not of particular types.

Sections 4–6 have been dropped for FACS Newsletter

References

[AKKB99] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brueckner, editors. *Algebraic Foundations of System Specification*. Springer Verrlag, 1999.

[BW82] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.

[GM92] J. Goguen and J. Meseguer. Order sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[Kap79] D. Kapur. Specifications of Majster's traversable stack and Veloso's traversable stack. *SIGPLAN Notices*, 14:46–53, May 1979.

[Ore79] F. Orejas. On the power of conditional specifications. *SIGPLAN Notices*, 14:78–81, July 1979.

[TWW82] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: parameterization and the power of specification techniques. *ACM TOPLAS*, 4:711–732, 1982.

Nineteenth International School for Computer Science Researchers

"Advances in Software Engineering"

Lipari Island

July 8 – July 21, 2007

The Nineteenth International School for Computer Science Researchers addresses PhD students and young researchers who want to get exposed to the forefront of research activity in the field of **Software Engineering**. The school will be held in the beautiful surroundings of the Island of Lipari.

Courses

- **First week**
  - Domain Analysis
    Dines Bjørner, *Technical University of Denmark, DK*
  - Problem Frames and their Composition
    Michael Jackson, *London, UK*
  - Software Safety
    Nancy Leveson, *MIT, USA*
  - Designing Evolvable Software Products
    Peter Sestoft, *Department of Natural Sciences, Royal Veterinary and Agricultural University,DK*
- **Second week**
  - Web Services
    Boualem Benatallah, *The University of New South Wales,Australia*
  - High-Level Modeling Patterns
    Egon Boerger, *University of Pisa, Italy*
  - Cooperation in Internet-Wide Environments
    Carlo Ghezzi, *Politecnico di Milano,Italy*
  - Security for Distributed Software
    Dieter Gollmann, *Hamburg-Harburg (TUHH), Germany*

Participants will be arranged in a comfortable hotel at very special rates. The conference room (in the same hotel) is air-conditioned and equipped with all conference materials. Special areas are reserved to students for the afternoon coursework and study. The island of Lipari can be easily reached from Milazzo, Palermo, Naples, Messina and Reggio Calabria by ferry or hydrofoil (50 minutes from Milazzo).

Two kinds of participant are welcome. Students: Participants who are expected to do afternoon courseworks and take a final exam. Auditors: Participants who are not interested in taking the final exam.

Up to 60 students and a limited number of additional auditors will be admitted. Deadline for application is **March 31, 2007**.

Applicants must include short curriculum vitae and specify two professors whom letters of recommendation will be asked to, if deemed necessary.

Applicants will be notified about admission by **April 14, 2007**. Registration fee is 650 Euro. The fee covers the course material (memory stick plus one book), bus+hydrofoil Catania airport-Lipari-Catania airport, 2 social tours (Salina and Panarea) and the social dinner on the beach of Vulcano. Late registration is 750 Euros. For details regarding registration, travelling information and other logistical information please consult the school web site **http://lipari.cs.unict.it**
The official language is English.

Directors

Alfredo Ferro	**Egon Börger**
University of Catania (Co-Chair)	University of Pisa (Co-Chair)

Prof. Alfredo Ferro
Dipartimento di Matematica e Informatica
Città Universitaria
Viale A. Doria, 6
95125 CATANIA
ITALY

Voice: +39-095- 7383071
Fax: +39-095-7337032
e-mail: ferro@cs.unict.it

Understanding the differences between VDM and Z

I. J. Hayes & C.B. Jones & J.E. Nicholls

I. J. Hayes*
e-mail: Ian.Hayes@itee.uq.edu.au
School of ITEE, University of Queensland,
Brisbane, 4072, Australia

C. B. Jones
e-mail: cliff@cs.ncl.ac.uk
Department of Computer Science, University of Newcastle upon Tyne,
Newcastle, U.K.

J. E. Nicholls
Oxford University Computing Laboratory
11 Keble Road, Oxford, OX1 3QD, U.K.

Abstract

This paper attempts to provide an understanding of the interesting differences between two well-known specification languages.

Introduction

The main ideas are presented in the form of a discussion. This was partly prompted by Lakatos's book 'Proof and Refutations' but, since this paper is less profound, characters from the children's television series 'The Magic Roundabout' are the speakers: Zebedee speaks for Z, Dougal puts the VDM position, and Florence acts as the user.

The specifications which are presented have been made similar so as to afford comparison – in neither the VDM nor the Z case would they be considered to be ideal presentations. Some technical details are relegated to footnotes.

Discussion

Florence: I know that some people are confused by the existence of two specification languages, Z and VDM, which have a lot in common.

Dougal: Yes, there is certainly a common objective in Z and VDM and in many places one can see that the same solution has been adopted.

*Copyright ©1993. All rights reserved. Reproduction of all or part of this work is permitted for educational or research purposes on condition that (1) this copyright notice is included, (2) proper attribution to the author or authors is made and (3) no commercial gain is involved.

This paper is available as a technical report UMCS–93–8–1 from the University of Manchester and has also been published in the *FACS Europe* newsletter, Autumn, 1993. Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

The use of both VDM and Z has been concentrated on the specification of *abstract machines*, and they both take the same so called 'model-oriented' approach. Jointly they differ from the so called 'algebraic' specification languages (these might be better called 'property-oriented' to contrast with 'model-oriented') which concentrate on specifying abstract data types.

Zebedee: The primary difference between these approaches is that VDM and Z both give an explicit model of the state of an abstract machine – the operations of the abstract machine are defined in terms of this state – whereas 'algebraic' approaches give no explicit model of the type – an abstract data type is specified by axioms giving relationships between its operations. For (a much-overused) example, a stack in VDM and Z would typically be modelled as a sequence, while in the 'algebraic' approaches axioms such as

$$pop(push(x, s)) = s$$

would be given.

Dougal: Before going further, let's be clear what we mean by 'VDM'. Strictly, VDM was always seen as a development method in the sense that rules are given to verify steps of development (data reification and operation decomposition). I guess that our discussion today will be confined to the specification language used in VDM.

Florence: Does that have a name?

Dougal: I wish I could say 'no'! But I have to confess that it is sometimes known as 'Meta-IV' – the draft VDM standard talks about 'VDM-SL',[1] but let's talk about the significant differences between the two specification languages.

Florence: Probably what hit me first about the difference between VDM and Z specifications is the appearance of the page. VDM specifications are full of keywords and Z specifications are all 'boxes'. Is this a significant difference?

Dougal: VDM uses keywords in order to distinguish the roles of different parts of a specification. For example, a module corresponds to an abstract machine with state and operations; a state has components and an invariant; and operations have separate pre- and post-conditions. These different components are distinguished as they have different purposes in the specification. To do this VDM makes use of keywords to introduce each component. All this structure is part of the VDM language.

Zebedee: In Z almost none of the structure Dougal mentioned is explicit in the language. A typical specification consists of lots of definitions, many of which are *schemas* – the boxes Florence was referring to. Schemas are used to describe not only states but also operations. The status of any particular schema is really only determined by the text introducing it, although it isn't hard to guess the purpose of a schema by looking at its definition. Schemas are also used as building blocks to construct descriptions of machine states or facets of operations. Such component building blocks may not correspond to any of the VDM categories.[2]

Florence: But don't you need the extra structure that VDM gives if you want to do formal refinements?

Zebedee: Yes, but to perform refinements you also need to consider a target programming language. For example, if you wanted to produce Modula code then you could give a definition module for an abstract machine in which the state and the operations are defined by Z schemas. Such a definition module would be very close to a VDM module in terms of structure.

Florence: How about an example – something other than stacks please!

[1]The notation used here is close to that of the 'Committee Draft' of the VDM-SL Standard but there may be minor syntactic differences.

[2]It is perhaps worth explaining that Z has a number of levels: a basic mathematical notation (similar to that of VDM); the schema notation; and conventions for describing the state and operations of an abstract machine using Z schemas. Little new notation is introduced in the third level, only conventions for making use of the other notation to describe abstract machines. It is worth noting that Z notation – the first two levels – has been used with a different set of conventions for other purposes, such as specifying real-time systems.

Zebedee: A simple relational database, known as NDB, has been presented in both notations. Let's use that.[3] After you Dougal.

Dougal: Someone writing a VDM specification of a system normally sketches a state before going into the details of the operations which use and modify that state.[4]

For the NDB description, the overall state will have information about entities and relations. In order to build up such a state, a number of sets are taken as basic.

Eid Each entity in the database has a unique identifier taken from the set *Eid*.

Value Entities have values taken from the set *Value*.

Esetnm An entity can belong to one or more entity sets (or types). The names of these sets are taken from the set *Esetnm*.

Rnm The database consists of a number of relations with names taken from the set *Rnm*.

All but the first of these can conveniently be thought of as parameters to the specification of the NDB database system; *Eid* is an internal set about which we need to know little – we just regard it as a set of 'tokens'.

Now we can begin to think about the types which are constructed from these basic sets. NDB is a binary relational database consisting of relations containing tuples which each have two elements: a *from* value and a *to* value. A tuple contains a pair of *Eid*s; in VDM the type *Tuple* is defined as follows

$$\begin{aligned} \mathit{Tuple} :: \mathit{fv} &: \mathit{Eid} \\ \mathit{tv} &: \mathit{Eid} \end{aligned}$$

Then a relation can be defined as a set of such pairs:

$$\mathit{Relation} = \mathit{Tuple}\text{-set}$$

To define whether a relation is to be constrained to be one-to-one – or whatever – four distinct constants are used

$$\mathit{Maptp} = \textsc{OneOne} \mid \textsc{OneMany} \mid \textsc{ManyOne} \mid \textsc{ManyMany}$$

Relation information (*Rinf*) contains information stored for a relation: apart from the *Tuple* set, an element of *Maptp* provides the constraint on the form of relation allowed. The consistency of the *tp* and *r* fields of *Rinf* is expressed as an invariant.

$$\begin{aligned} \mathit{Rinf} :: \mathit{tp} &: \mathit{Maptp} \\ r &: \mathit{Relation} \end{aligned}$$

$$\begin{aligned} &\mathsf{inv}\ (\textit{mk-Rinf}(tp, r)) \triangleq \\ &\quad (tp = \textsc{OneMany} \Rightarrow \forall t_1, t_2 \in r \cdot t_1.tv = t_2.tv \Rightarrow t_1.fv = t_2.fv) \wedge \\ &\quad (tp = \textsc{ManyOne} \Rightarrow \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \Rightarrow t_1.tv = t_2.tv) \wedge \\ &\quad (tp = \textsc{OneOne} \Rightarrow \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \Leftrightarrow t_1.tv = t_2.tv) \end{aligned}$$

It's worth noting that the set of values defined by a definition with an invariant only contains values which satisfy the invariant. So we can only say that $rel \in Rinf$ if the relation, $rel.r$ is consistent with the map type $rel.tp$. Because invariants can be arbitrary predicates, type membership is only partially decidable.

Florence: Is Z's notion of type the same as what Dougal just described?

[3] Originally formally specified in [Wel82] and revised in [Wal90], this was used as a challenge problem in [FJ90] to which a Z response is given in [Hay92].

[4] The description of the NDB state presented here is given *postfacto* rather than attempting to emulate the process by which specifications are produced.

Zebedee: No, but this is a difference in the use of the word 'type' rather than a real difference between Z and VDM. In Z the term 'type' is used to refer to what can be statically type checked. This is more liberal than what Z calls a 'declared set' which is what VDM calls a 'type'.

Dougal: Well, let me get through the rest of the state; then we can make more comparisons.

In NDB, relations are identified not only by their names but also by the entity sets of the values they relate (so a database can contain two relations called Owns: one between People and Cars and – at the same time – another between People and Houses). So a key for a relation contains three things

$$\begin{aligned} Rkey :: \ & nm : Rnm \\ & fs : Esetnm \\ & ts : Esetnm \end{aligned}$$

The overall state which we are aiming for has three components: an entity set map (*esm*) which defines which entities are in each valid entity set; a map (*em*) which contains the value of each identified entity; and a third map (*rm*) which stores the relevant *Rinf* for each *Rkey*. The invariant records the consistency conditions between the components.

$$\begin{aligned} Ndb :: \ & esm : Esetnm \xrightarrow{m} Eid\text{-set} \\ & em : Eid \xrightarrow{m} Value \\ & rm : Rkey \xrightarrow{m} Rinf \end{aligned}$$

$$\begin{aligned} & \mathsf{inv}\ (mk\text{-}Ndb(esm, em, rm)) \triangleq \\ & \quad \mathsf{dom}\ em = \bigcup \mathsf{rng}\ esm \land \\ & \quad \forall\, rk \in \mathsf{dom}\ rm \cdot \{rk.fs, rk.ts\} \subseteq \mathsf{dom}\ esm \land \\ & \qquad\qquad\qquad \forall\, mk\text{-}Tuple(fv, tv) \in rm(rk).r \cdot fv \in esm(fs) \land tv \in esm(ts) \end{aligned}$$

Later, we'll look at how this (together with the initial state and the operations) gets grouped into a module.

Florence: How would the above state be presented in Z?

Zebedee: All the details would be almost identical, but the specification would be structured differently. The specification would consist of a sequence of sections and each section would present a small set of the state components along with operations on just those components.

The Z approach to structuring specifications is to try to build the specification from near orthogonal components. We look for ways of splitting the state of the system so that we can specify operations on just that part of the state that they require.

For NDB we have chosen to split the specification into three parts:

1. entities and their types or entity sets,
2. a single relation, and
3. multiple relations.

Finally, we put these specifications together to give the final specification.

Rather than follow the normal Z approach here, I'll give all of the state components from the different sections together, so that we can compare the state with that used for the VDM specification. As for the VDM, our basic sets are the following:

$$[Eid, Esetnm, Value]$$

As with the VDM the sets *Esetnm* and *Value* can be thought of as parameters, and the set *Eid* is used locally within the specification.

For a database we keep track of the entities that are in an entity set (of that type). Every entity must be of one or more known types. The following schema, *Entities*, groups the components of the state together with a invariant linking them.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Entities}}\\ \hline
esm : Esetnm \mathrel{+\!\!\!\!\twoheadrightarrow} (\mathbb{F}\ Eid)\\
em : Eid \mathrel{+\!\!\!\!\twoheadrightarrow} Value\\ \hline
\operatorname{dom} em = \bigcup \operatorname{ran} esm\\ \hline
\end{array}$$

Florence: How does this differ from the VDM so far?

Zebedee: It's virtually identical – bar the concrete syntax. The notation $\mathbb{F}\ Eid$ is equivalent to the VDM notation Eid-set.

Another approach to defining the state in Z would be to define *esm* as a binary relation between *Esetnm* and *Eid*. This leads to simpler predicates in the specifications because the Z operators on binary relations are closer to the operations required for NDB's binary relations. *Entities* is defined using Z's binary relations in [Hay92], but for our comparison here it is simpler to use the same state as the VDM version. That way we can concentrate on more fundamental differences.

Dougal: Yes, these modelling differences are interesting but not the point of today's discussion; but it is worth saying that binary relation notation could also be added to VDM and the same alternative state used.

Florence: So far that only covers the components *esm* and *em* of the VDM *Ndb* state.

Zebedee: Right, and at this point we would give a set of operations on the above state, but in order to provide a more straightforward comparison with the VDM state, we shall skip to the remainder of the state.

The relations used in NDB are binary relations between entity identifiers (rather than entity values). A *Tuple* is just a pair of entity identifiers and a *Relation* is modeled as a Z binary relation between entity identifiers.

$$Tuple == Eid \times Eid$$
$$Relation == Eid \leftrightarrow Eid$$

Florence: Is that really different from VDM?

Zebedee: Well, yes and no. Both the VDM and Z versions use a set of pairs for a relation – note that $Eid \leftrightarrow Eid$ is a shorthand for $\mathbb{P}(Eid \times Eid)$. The difference is that, in Z, relations are predefined and have a rich set of operators defined on them.

Dougal: There are a couple of points I'd like to pick up from what Zebedee has said. When we make a selection of basic building blocks for specifications, we are clearly influenced by experience. There is nothing deep in, say, the omission of relations from VDM (or – say of optional objects from Z). Once again, the remarkable thing is just how similar the selection in Z and VDM is. As I said above, if I were writing a large specification which needed relations, I would just extend VDM appropriately.

Florence: What about the *Maptp* in Z?

Zebedee: *Maptp* is virtually identical:

$$Maptp ::= OneOne \mid OneMany \mid ManyOne \mid ManyMany$$

When a relation is created its type is specified as being one of the following four possibilities: it is a one-to-one relation (i.e., an injective partial function), a one-to-many relation (i.e., its inverse is a partial function), a many-to-one relation (i.e., a partial function), or a many-to-many relation. In Z, the set of binary relations between X and Y is written $X \leftrightarrow Y$, the set of partial functions is written $X \rightarrowtail\!\!\!\!\!+ Y$, the set of one-to-one partial functions is written $X \rightarrowtail\!\!\!\!\!+\!\!\!\!\!\rightarrowtail Y$, and the inverse of a relation r is written $r^{\sim}$.

A relation is created to be of a particular type and no operation on the relation may violate the type constraint.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rinf}}\\ \hline
tp : \mathit{Maptp}\\
r : \mathit{Relation}\\ \hline
(tp = \mathit{OneOne} \Rightarrow r \in \mathit{Eid} \pinj \mathit{Eid}) \land\\
(tp = \mathit{ManyOne} \Rightarrow r \in \mathit{Eid} \pfun \mathit{Eid}) \land\\
(tp = \mathit{OneMany} \Rightarrow r^{\sim} \in \mathit{Eid} \pfun \mathit{Eid})\\ \hline
\end{array}$$

Dougal: If you expanded the definitions you would get the same constraint as in the VDM version.

Zebedee: Yes, they are exactly the same.

Florence: We still don't have NDB's named relations in Z.

Zebedee: That's next, but again at this point in the normal flow of a Z specification operations would be defined on the state *Rinf*. The database consists of a number of relations with names taken from the set *Rnm* (essentially a parameter set).

$[\mathit{Rnm}]$

A relation is identified by its name and the 'from' and 'to' entity sets that it relates. This allows a number of relations to have the same *Rnm* provided they have different combinations of 'from' and 'to' entity sets.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rkey}}\\ \hline
nm : \mathit{Rnm}\\
fs, ts : \mathit{Esetnm}\\ \hline
\end{array}$$

The entities related by each relation must belong to the entity sets specified by the relation key.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ndb}}\\ \hline
\mathit{Entities}\\
rm : \mathit{Rkey} \ffun \mathit{Rinf}\\ \hline
\forall rk : \mathrm{dom}\, rm \bullet\\
\qquad \{rk.fs, rk.ts\} \subseteq \mathrm{dom}\, esm \land\\
\qquad (rm\ rk).r \in esm(rk.fs) \leftrightarrow esm(rk.ts)\\ \hline
\end{array}$$

Florence: Because you have included *Entities* you now have all the state components and the invariant from *Entities*, so that this is equivalent to the VDM *Ndb* state.

Zebedee: Yes.

Florence: Why don't we look at initialisation?

Dougal: The initial state (in this case it is unique) is defined in VDM as

$$\mathsf{init}\ (ndb) \triangleq ndb = \mathit{mk\text{-}Ndb}(\{\mapsto\}, \{\mapsto\}, \{\mapsto\})$$

Zebedee: In Z the initial state is defined by the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ndb\_Init}}\\ \hline
\mathit{Ndb}\\ \hline
esm = \{\} \land em = \{\} \land rm = \{\}\\ \hline
\end{array}$$

Again, if we followed the more structured presentation of NDB, we would probably define the set of allowable initial *Entities* and then define the allowable initial *Ndb* states in terms of it.

Florence: Why don't we look at operations?

Dougal: The simplest operation adds a new entity set name to the set of known entity sets, *esm*. The set of entities associated with this new name is initially empty. In VDM this can be defined.

$$
\begin{array}{l}
ADDES\,(es : Esetnm)\\
\mathsf{ext\ wr}\ esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}\\
\mathsf{pre}\ \ es \notin \mathsf{dom}\, esm\\
\mathsf{post}\ esm = \overleftarrow{esm} \cup \{es \mapsto \{\,\}\}
\end{array}
$$

Zebedee: In Z that's

$$
\begin{array}{|l}
\hline
ADDES0\\
\Delta Entities\\
es? : Esetnm\\
\hline
es? \notin \mathrm{dom}\, esm \land\\
esm' = esm \cup \{es \mapsto \{\}\} \land\\
em' = em\\
\hline
\end{array}
$$

where *ΔEntities* introduces the *before* and *after* (primed) states:

$$\Delta Entities \mathrel{\widehat{=}} Entities \land Entities'$$

Florence: One obvious syntactic difference is that VDM uses hooked variables for the *before* state and unhooked variables for the *after* state, whilst Z uses undecorated variables for the *before* state and primed variables for the *after* state. In addition, Z uses variable names ending in '?' for inputs (and names ending in '!' for outputs). Apart from the differences in syntax, I notice that VDM uses an externals clause and distinguishes the pre-condition.

Zebedee: Yes, Z does not have any equivalent of an externals clause. The predicate must define the final values of all variables, even if the variable is unchanged, such as *em*. This is why in Z one divides up the state into small groups of components and defines sub-operations on each group, before combining the sub-operations in order to define the full operation. For a large specification with many state components, if one had to define the operations on the whole state, then there would be many boring predicates stating that many of the variables are unaffected. Dividing up the state avoids this problem, although it is still necessary to promote the operations on the substates to the full state at some stage.

Dougal: In VDM, an operation is always written in a module. This provides the appropriate state but one can use the external clause of an operation specification to make it self-contained and to restrict the frame.

Zebedee: In Z, the state is explicitly included – via a '*Δ*' or '*Ξ*' schema usually – within the operation, so the operation schema can stand on its own.

With regard to the pre-condition, although the same predicate appears in the Z schema *ADDES*0 as in the VDM operation *ADDES*, it is not separated out. For this operation that doesn't make a large difference between the Z and VDM versions, but for other operations it can.

Dougal: OK, let's look at deleting an entity set in VDM.

$$
\begin{array}{l}
DELES\,(es : Esetnm)\\
\mathsf{ext\ wr}\ esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}\\
\qquad \mathsf{rd}\ \ rm : Rkey \xrightarrow{m} Rinf\\
\mathsf{pre}\ \ es \in \mathsf{dom}\, esm \land esm(es) = \{\,\} \land\\
\qquad \forall\, rk \in \mathsf{dom}\, rm \cdot es \neq rk.fs \land es \neq rk.ts\\
\mathsf{post}\ esm = \{es\} \mathbin{\lhd\!\!\!-} \overleftarrow{esm}
\end{array}
$$

Zebedee: In Z this would be written

$$\begin{array}{|l}
\hline
\mathit{DELES0} \\
\Delta \mathit{Entities} \\
es? : \mathit{Esetnm} \\
\hline
es? \in \mathrm{dom}\, esm \land esm(es) = \{\} \land \\
esm' = \{es?\} \ntriangleleft esm \land \\
em' = em \\
\hline
\end{array}$$

Florence: But isn't it missing part of the pre-condition in the VDM version?

Zebedee: Yes, *DELES*0 is only defined on the state *Entities*, so it is impossible to talk about the state component *rm*. To define the equivalent of the VDM operation, we need to promote *DELES*0 to the full *Ndb* state. We do this by defining the schema *ΞRM* which introduces the full *Ndb* state and constrains *rm* to be unchanged. This is then conjoined with *DELES*0.

$$\begin{array}{l}
\Xi RM \mathrel{\widehat{=}} [\Delta Ndb \mid rm' = rm] \\
DELES \mathrel{\widehat{=}} DELES0 \land \Xi RM
\end{array}$$

Florence: But I still can't see the missing bit of the pre-condition!

Zebedee: That's because it's not visible! But the Z *DELES* has the same pre-condition as the VDM operation.

Florence: How do I get to see it?

Zebedee: In Z, the pre-condition of an operation characterises exactly those inputs and initial states such that there exists a least one possible combination of outputs and final state that satisfies the operation specification. For *DELES* the pre-condition is

$$\begin{array}{|l}
\hline
\mathrm{pre}\ \mathit{DELES} \\
\mathit{Ndb} \\
es? : \mathit{Esetnm} \\
\hline
\exists\, Ndb' \bullet \\
\quad es? \in \mathrm{dom}\, esm \land esm(es) = \{\} \land \\
\quad esm' = \{es?\} \ntriangleleft esm \land \\
\quad em' = em \land \\
\quad rm' = rm \\
\hline
\end{array}$$

The predicate can be expanded to

$$\begin{array}{|l}
\hline
\mathrm{pre}\ \mathit{DELES} \\
\mathit{Ndb} \\
es? : \mathit{Esetnm} \\
\hline
\exists\, Entities'; rm' : Rkey \pinj Rinf \bullet \\
\quad (\forall\, rk : \mathrm{dom}\, rm' \bullet \\
\qquad \{rk.fs, rk.ts\} \subseteq \mathrm{dom}\, esm' \land \\
\qquad (rm'\ rk).r \in esm'(rk.fs) \leftrightarrow esm'(rk.ts) \land \\
\quad es? \in \mathrm{dom}\, esm \land esm(es) = \{\} \land \\
\quad esm' = \{es?\} \ntriangleleft esm \land \\
\quad em' = em \land \\
\quad rm' = rm \\
\hline
\end{array}$$

which can be simplified to

$$\begin{array}{|l}
\text{pre } \mathit{DELES} \\
\mathit{Ndb} \\
es? : \mathit{Esetnm} \\
\hline
es? \in \mathrm{dom}\, esm \land esm(es) = \{\} \land \\
(\forall\, rk : \mathrm{dom}\, rm \bullet es? \neq rk.fs \land es? \neq rk.ts) \\
\hline
\end{array}$$

Dougal: That's now the same as the VDM pre-condition.

Florence: Wasn't all that a bit complicated compared to the VDM version?

Zebedee: Well, yes and no. If you want to compare it with the VDM version, then we have also done the equivalent of discharging its *satisfiability* proof obligation. In VDM this proof obligation is the main consistency check available for the specifier, and the Z pre-condition calculation can be likewise seen as a consistency check – that the calculated pre-condition agrees with the specifier's expectations.

Dougal: I believe that this *is* a significant difference between Z and VDM. There is a technical point: when development steps are undertaken, the pre-condition is required.[5] But there is also a pragmatic point: in reading many industrial (informal) specifications, I have observed that people are actually not so bad at describing what function is to be performed; what they so often forget is to record the assumptions. I therefore think that it is wise to prompt a specifier to think about the pre-condition.

Zebedee: I'd agree with that, but in both VDM and Z, provided the respective consistency checks are done, we do end up at the same point. It's only the path that is different.

With the Z approach of constructing the specification of an operation it is only when you have the final operation that the concept of a pre-condition really makes sense.[6]

Florence: What does a pre-condition mean? I'm really not clear about whose responsibility it is to avoid calls that violate the pre-condition – or are these exceptions?

Dougal: A pre-condition is essentially a warning to the user: the behaviour defined in the post-condition is only guaranteed if the starting state satisfies the pre-condition. In formal development, the user should treat the pre-condition as a proof obligation that an operation is only invoked when its pre-condition is true. It is perhaps useful to think of the pre-condition as something that the *developer* of an operation can rely upon; it is permission for the developer to ignore certain situations. Exceptions are quite different – VDM does have notation (not used so far in this example) to mark error conditions which must be detected and handled by the developer. In VDM an operation specification can consist of a normal pre/postcondition pair followed by a set of exception pre/postcondition pairs.

Florence: Can we compare the treatment of exceptions? I've noticed Z doesn't have exceptions as part of the language.

Zebedee: I guess one way of viewing the Z approach is to say that it doesn't really have exceptions at all. As part of specifying an operation one specifies its behaviour both in the normal case and in the exceptional cases. Both the normal case and each exceptional case are specified in the same manner in a Z schema. Each of these schemas corresponds to a pre/post-condition pair in the VDM version.

Florence: So the difference about the use of pre/post-condition pairs in the VDM version versus a single schema in the Z version crops up here as well.

Zebedee: That's correct.

Florence: How else do they differ?

[5] Tony Hoare in [Hoa91] appears to argue for the use of Z to develop specifications and the use of VDM for development of the design and implementation.

[6] Although it is possible to define operators similar to schema conjunction and disjunction on pre/postcondition pairs; see [War93].

Zebedee: In terms of what they both mean, not at all. The complete operation is specified in Z by taking the disjunction of the schemas for the normal and exceptional cases. It has the same meaning as the corresponding VDM specification. For example, in both Z and VDM the pre-conditions of the alternatives may overlap, either between the normal case and an error case or between error alternatives, and in both Z and VDM there is a nondeterministic choice between alternatives that overlap.

Dougal: Yes, that's correct. Although VDM and Z appear to describe exceptions differently, the semantic ideas underneath the concrete syntax are virtually identical.

Florence: So why do they look so different?

Zebedee: Well mostly it is just differences in syntax but I guess there are a couple of points about building an operation specification from Z schemas using schema operators that are worth noting. Firstly, it is possible to specify more than one normal case and these further alternatives just become part of the disjunction of cases. (There is no real distinction between normal and error cases, so one can have as many of either as suits the problem in hand.) Secondly, the same *exception* schema can be used for more than one operation. This has the twin advantages of avoiding repetition and maintaining consistency between different operations in their treatment of the same exception.

Dougal: I see those advantages; it is just in the spirit of VDM to have a place for exceptions marked by keywords. As always, syntactic issues tend to be more an issue of taste than of hard scientific arguments.

Florence: How about an example?

Zebedee: Let's consider the possibility of trying to add a new entity set when the name is already in use. In Z the exception alternative is specified by

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ESInUse}} \\
\hline
\Xi \mathit{Entities} \\
es? : \mathit{Esetnm} \\
\hline
es? \in \operatorname{dom} esm \\
\hline
\end{array}$$

where $\Xi Entities$ introduces the *before* and *after* states and constrains them to be equal:

$$\Xi Entities \mathrel{\widehat=} [\Delta Entities \mid \theta Entities' = \theta Entities]$$

The operation augmented with the error alternative is

$$ADDESX \mathrel{\widehat=} ADDES0 \lor ESInUse$$

Dougal: In VDM *ADDES* with an exception if the entity set is already in use would be specified by adding the line

$$\mathsf{errs}\ ESInUse : es \in \mathsf{dom}\ esm$$

Zebedee: In both the VDM and Z we should really add an error report to be returned by the operation. This would be done in essentially the same way in both VDM and Z.

Dougal: In VDM, the whole specification gets put into a *module*: this is a structuring mechanism that makes it possible to build one module on top of others. One possible module syntax is illustrated in Appendix A. But I have to confess that this is still a subject of debate. In fact [FJ90] was written precisely because this debate is not yet settled.

Zebedee: Z also needs a modularisation mechanism and one proposal is developed in [HW93].

Florence: Does the issue of pre-conditions have any connection with the fact that I suspect the two notations handle partial functions differently?

Dougal: Yes, there is a loose connection and there are differences between Z and VDM here. In fact, I'll be interested to hear what Zebedee has to say on this point.

Let me set the scene.[7] Both operators like hd and recursively defined functions can be partial in that a simple type restriction does not indicate whether they will evaluate to a defined result for all arguments of the argument type: $\mathsf{hd}\,[\,]$ or factorial applied to minus one are examples of non-denoting terms. VDM uses non-standard logical operators which cope with possibly undefined operands: so $\mathsf{true} \vee \mathit{undefined}$ is true as is $\mathit{undefined} \vee \mathsf{true}$.

Zebedee: In [Spi88] Mike Spivey uses existential equality which always delivers a truth value – where either of the operands are undefined, the whole expression is false. This enables him to stay with classical (two-valued) logic.

Dougal: Yes, I should have said that there are a couple of different approaches where one tries to trap the undefined before it becomes an operand of a logical operator. Thus, in $\mathsf{hd}\,[\,] =_{\exists} 5$ it is possible to define the result as false. Unfortunately, the task does not end here – any relational operators need similar special versions. Moreover, the user has to keep the distinction between the logical equality and the computational equality operator of the programming language in mind. As they say 'There ain't no such thing as a free lunch'.

Zebedee: Yes, but I did say Spivey took that approach; in the beginning, I believe that Jean-Raymond Abrial wanted to formalize the view that while 'the law of the excluded middle' held, for undefined formulae, one never knew which disjunct was true. (The work on the new Z standard is still evolving.)

Florence: This is all a bit technical – does it matter?

Zebedee: Not much – but it is an interesting difference!

Florence: What about recursively defined structures such as trees?

Zebedee: Before we start into the details of the definition of recursive structures, one approach often taken in Z specifications is to avoid recursive structures and use a flattened representation instead. For example, the specification of the Unix filing system [MS84] represents the hierarchical directory structure by a mapping from full path names to the file's contents. All prefixes of any path name in the domain of the map must also be in the domain of the map.

A representation of the filing system state[8] as a recursive structure can be defined by considering a directory to be a mapping from a single segment of a path name to a file, where a file is either a sequence of bytes or is itself a directory.

It is interesting to compare specifications of filing system operations on both representations. On the flat representation finding a file is simply application of the map representing the filing system state to the file name, whereas a recursive function needs to be provided for the recursive representation. If one considers updating operations, the flat representation provides even simpler descriptions of operations.

Dougal: The recursive approach description is used in [Jon90] and I'm not sure that I'd concede the advantages that Zebedee claims for the flat specification. But that's a modelling issue not a difference between the two specification languages.

Florence: So when do we need to use recursive structures?

Dougal: A good example is the specification of the abstract syntax of a programming language.

Florence: Are there examples outside the rather special field of programming languages?

Zebedee: Yes, consider a simple binary tree. This can be specified in Z via the following

$$T ::= \mathit{nil} \mid \mathit{binnode}\langle\!\langle T \times \mathbb{N} \times T \rangle\!\rangle$$

This introduces a new type T, a constant of that type nil and an (injective) constructor function $\mathit{binnode}$, that when given a left sub-tree, a natural number and a right sub-tree returns a tree.

[7] A fuller discussion can be found in [BCJ84, CJ91, JM93].

[8] We avoid consideration of inodes and links here.

Dougal: In VDM that would be

$$\begin{aligned} \mathit{binnode} :: l &: T \\ v &: \mathbb{N} \\ r &: T \end{aligned}$$

$$T = [\mathit{binnode}]$$

Zebedee: There are some technical differences in the approach taken here. In Z, T is a new type, whereas for the VDM *binnode* is a new type but T is not a new type. Also, in neither Spivey [Spi92] nor the Z Base Standard are mutually recursive structures (as used in the VDM version above) allowed.

Dougal: Wouldn't that cause problems for specifying the abstract syntax of a programming language? For an Algol-like language it is common to distinguish the syntactic categories of commands and blocks, but a command may be a block and a block may contain commands.

Zebedee: If you wanted to follow the draft Z standard then you would have to merge commands and blocks into a single syntactic category and then add consistency constraints to the language to ensure that they are correctly used.

Dougal: So, it is possible to specify it, but it isn't the most natural specification.

Zebedee: True. I have to admit that I would be very tempted to extend Z to allow mutually recursive definitions.[9]

Florence: I suppose this is all linked to the semantics of the specification languages themselves.

Zebedee: To the average user, the semantics of the meta-language might not be bedtime reading. The aim has always been to base the semantics of Z on set theory; Mike Spivey gives a semantics in [Spi88] but a new semantics is being developed for the language standard.

Dougal: VDM has its origins in language description and it has to cope with reflexive domains etc. The semantics in the 'Committee Draft' of the VDM-SL standard is certainly not 'bedtime reading' but for simple operations a relatively simple set theoretic semantics would suffice.

Florence: Could you each tell me a bit about the history of your chosen specification languages?

Dougal: VDM was developed at the IBM Laboratory in Vienna. The Laboratory came into existence in 1961 when Professor Heinz Zemanek of the Technical University in Vienna decided to move his whole group to an industrial home. They had previously developed a computer called Mailüfterl at the Technical University. From 1958 the group had been increasingly involved in software projects including the construction of one of the early compilers for the ALGOL 60 programming language. As time went on they found it difficult to get adequate support for their projects and eventually joined IBM. Still in the first half of the 1960s, IBM decided to develop a new programming language for which the ambition was to replace both FORTRAN and COBOL. The language, which was at first called New Programming Language (until the National Physical Laboratories in the UK objected to the acronym – the language became known as PL/I), was clearly going to be large and it was decided that it would be useful to try to apply some formal techniques to its description.

Based on their own work – and influenced by research work by Cal Elgot, Peter Landin and John McCarthy – the Vienna group developed an operational semantics definition of PL/I which they called ULD-3 (Universal Language Description; ULD-2 was the name which had been applied to the IBM Hursley contribution to this effort; the language itself was being developed mainly from Hursley along with the early compilers. ULD-1 was a term applied to the natural language description of the language.)[10] The description of PL/I in ULD-3 style ran through three versions. These are very large documents. Operational semantics is now seen as unnecessarily complicated

[9]The problem here is the scope of the definitions *vis a vis* constraints on the set defined by the recursive structure.

[10]VDL stands for Vienna Description Language and was a term coined by JAN Lee for the notation used in ULD-3.

when compared to denotational semantics. However, to make the principles of denotational semantics applicable to a language like PL/I with arbitrary transfer of control, procedures as arguments, complicated tasking, etc., required major theoretical break-throughs and a considerable mathematical apparatus not available at the time. The effort of the formal definition uncovered many language problems early and had a substantial influence on the shape of the language.

Towards the end of the 1960s serious attempts were made to use the ULD-3 description as the basis of compiler designs. Many problems were uncovered. The over-detailed mechanistic features of an operational semantics definition considerably complicated the task of proving that compiling algorithms were correct. But again one should be clear that an achievement was made; a series of papers was published which did describe how various programming language concepts could be mapped into implementation strategies which could be proved correct from the description. A series of proposals were made which could simplify the task of developing compilers from a semantic description. One of these was an early form of an exit construct which actually led to an interesting difference between the Vienna denotational semantics and that used in Oxford. Another important idea which arose at this time was Peter Lucas' *twin machine* proof and subsequently the observation that the ghost variable type treatment in the twin machine could be replaced by retrieve functions as a simpler way of proving that this sort of data development was correct. It is worth noting that Lucas' *twin machine* idea has been re-invented several times since: the generalization of retrieve functions to relations can be seen as equivalent to twin machines with invariants.

Hans Bekič initiated the move that pushed the Vienna group in the direction of denotational semantics; he spent some time in England with Peter Landin at Queen Mary College.

During the period from 1971 to 1973, the Vienna group was diverted into other activities not really related to formal description. Cliff Jones at this time went back to the Hursley Laboratory and worked on a *functional* language description and other aspects of what has become known as VDM. In particular he published a development of Earley's recogniser which is one of the first reports to use an idea of data refinement. In late 1972 and throughout '73 and '74 the Vienna group (Cliff Jones returned and Dines Bjørner was recruited) had the opportunity to work on a PL/I compiler for what was then a very novel machine. They of course decided to base their development for the compiler on a formal description of the programming language. PL/I at that time was undergoing ECMA/ANSI standardisation. The Vienna group chose to write a denotational semantics for PL/I. This is the origin of the VDM work. VDM stands for Vienna Development Method. When they decided not to go ahead with the machine, IBM decided to divert the Vienna Laboratory to other activities in 1976. This led to a diaspora of the serious scientists. Dines Bjørner went to Copenhagen University then on to the Technical University of Denmark. Peter Lucas left to join IBM Research in the States. Wolfgang Henhapl left to take up a chair in Germany and Cliff Jones left to move to IBM's European System Research Institute (ESRI) in Brussels.

Cliff Jones and Dines Bjørner took upon themselves the task of making sure that something other than technical reports existed to describe the work that had gone on on the language aspects of VDM. LNCS 61 ([BJ78]) is a description of that work. At ESRI, Cliff Jones also developed the work on those aspects of VDM not specifically related to compiler development and the first book on what is now generally thought of as VDM is [Jon80]. Both of these books have now been supplanted. The language description work is best accessed in [BJ82] and the non-language work is best seen in – second edition – [Jon90].

Dines Bjørner's group at the Technical University of Denmark strenuously pursued the use of VDM for language description and he and his colleagues were responsible for descriptions of the CHILL programming language and a major effort to document the semantics of the Ada programming language. Cliff Jones spent 1979 to 81 at Oxford University (collecting a somewhat belated doctorate). This was an interesting period because Cliff and Jean-Raymond Abrial arrived within a few days of each other in Oxford and had some interesting interchanges about the evolving description technique which through many generations has been known as Z.

The non-language aspects of VDM were taken up by the STL laboratory in Harlow and, partly because of their industrial push, BSI were persuaded to establish a standardisation activity. This activity has not been easy to get going because of the differences between the pressures of the language description aspects of VDM and those who are only interested in pre/post-conditions, data reification and operation decomposition. It is to the credit of the standards committee that they have managed to bear in mind the requirements of both sorts of user and come up with a standard which embraces such a wide scope of technical ideas. There are now many books on VDM and more papers than even Dines Bjørner's energy could keep in a bibliography although Peter Gorm-Larsen has made an attempt to continue the work of keeping the key references in a single bibliography [Lar93].

The ideas in VDM have influenced several other specification languages including RAISE, COLD-K and VVSL.

Florence: Can you tell me how Z started?

Zebedee: Well, although there was a Z notation before 1979, many people associate the early development of Z with the period spent by Jean-Raymond Abrial in Oxford from 1979 to 1981. Abrial had used a paper he had written with Steve Schuman and Bertrand Meyer as lecture notes for a course given in Belfast. He was invited by Tony Hoare to Oxford, and presented similar material to the Programming Research Group (PRG) where it generated considerable interest and resulting activity. The notation described in this paper includes a basis in set theory and predicate calculus, although at this time the schema notation had not been fully developed.

Jean-Raymond Abrial was in Oxford at the same time as Cliff Jones, who had already worked on the Vienna Definition Language and the Vienna Development Method. The intention was that the two should exchange ideas and objectives and there were productive communications between the two, although in the end each pursued a distinctive path.

Florence: Is there such a thing as a Z method?

Zebedee: Z is a *notation*, and there is no official method attached to it, though there are conventions and practices that make it specially suitable for specifications written in the model-oriented style. The status of Z as a mathematical notation (rather than a method) is deliberate, and gives it flexibility and open-endedness.

Florence: How did the notation develop after the first proposals?

Zebedee: As with much of the PRG research, early development of Z centred on industrial case studies. An early case study was CAVIAR, a visitor information system for an industrial company based on requirements from STL; other case studies carried out in the early stages included those based on the UNIX File System, the ICL Data Dictionary, and several on topics in Distributed Computing. PRG members carrying out case studies included Carroll Morgan, Ian Hayes, Bernard Sufrin, Ib Sørensen, and others. Ian Hayes, in addition to his contributions to the IBM CICS project, later collected these case studies and published them in 1987 in the first book on Z (second edition [Hay93]).

One of the most extensive case studies has been the use of Z for defining CICS, a transaction processing system developed by IBM. The collaboration between the PRG and the Hursley development laboratory, starting in 1982 and still continuing, has been a valuable source of information and experience for both groups.

During this early period the design of the most distinctive feature of Z, the *schema*, together with related schema operations, emerged in its present form. The Z schema notation was originally introduced as a technique for structuring large specifications and was seen as a means of naming and copying segments of mathematical text, much like a textual macro language. It was later apparent that schemas could be used more generally to define the combination of specifications, and the basic operations of schema inclusion and conjunction were extended to form the more comprehensive operations that make up what has been called the *schema calculus*.

Florence: You've talked about the PRG contribution to application case studies – what about the underlying theory?

Zebedee: In early stages of Z development, the notation was described in documents produced in the PRG and locally distributed. The complete language description, *The Z Handbook* by

B.A. Sufrin [Suf88], was given only a limited circulation, and in fact the first account of the notation published in book form was in the 1987 edition of the collection of *Case Studies* (second edition [Hay93]) mentioned above. Theoretical work on the foundations of Z continued in the PRG and elsewhere, and an important contribution was provided by the D.Phil. thesis of Mike Spivey, subsequently published as a book [Spi88].

With a growing number of industrial users of Z, requests for standardisation were made at Z User Meetings in 1987 and 1988. Work was started in the Programming Research Group to establish an agreed definition of the language. Starting with the best available documentation, including [Suf88], the document produced in 1989 as a result of this work, the *Reference Manual* by J.M. Spivey became a widely accepted description of Z and provided the main starting point for the standards work described below – it is now in its second edition [Spi92].

Florence: I believe there is now a draft standard for Z – what is the status of this?

Zebedee: Towards the end of 1989 a project to develop Standards, Methods and Tools for Z was set up, with supporting funding from the UK Department of Trade and Industry. The formation of the ZIP project marked the beginning of a further stage of development, providing a stable basis for the development of national and international standards for Z. As with other projects of this kind, members of the project included both industrial and academic partners. The project was divided into four main working groups dealing with Standards, Methods and Tools – there was also a Foundations group providing theoretical support, mainly for the standards work.

The Z Standard Group developed new material for the standard, not only providing a newly written document in the style needed for a standard, but also introducing new material for the semantics (see for example [GLW91]) and logic [WB92] defined in the standard. The first draft, Version 1.0 (reference) was presented at the Z Users Meeting in December 1992 and the standards committee is now at work, reviewing and revising the document as it becomes ready for standardisation in ISO.

Meanwhile, industry users are busy using Z on projects, writing tools for Z and considering how it can be combined with other notations and methods. A good idea of the breadth and variety of interest can be gained from the Z Bibliography [Bow92].

The standards committees for Z and VDM-SL keep in touch by exchange of documents and by the appointment of liaison members. They are both subcommittees of the same BSI standards committee.

Florence: Could you give me some useful references?

Zebedee: For Z, the standard reference for the language (until the language standard appears) is Mike Spivey's [Spi92]. However, this is a language reference manual and there are some more introductory texts such as [PST91, Wor92] and a book of case studies [Hay93].

Dougal: For the non-compiler aspects of VDM, the standard reference has been [Jon90] and a case studies book is [JS90]; but [WH93] refers to Jones' book as 'austere' and either of [AI91, LBC90] might be more approachable. A good overview of VDM-SL is contained in [Daw91]; although there are several books on the language description and compiler development aspects of VDM, they haven't really come up very much in our discussion.

Florence: You have both ignored details of concrete syntax of the mathematical notation: these differences confuse some people.

Zebedee: Yes, but they are just an accident of history.

Dougal: A list of the syntactic differences has been given in a note [ISO91] from the Japanese ISO representatives.

Florence: Well, it's time for bed.

Zebedee: Boing!

Dougal: *Chases his tail for a bit before running off to bed.*

Acknowledgements

We would like to acknowledge the input from John Fitzgerald, and for permission to reuse the NDB material from [FJ90]. Peter Gorm Larsen, Lynn Marshall, Anthony Hall, Tony Hoare and Tim Clement gave us useful comments on drafts of this paper. Cliff Jones thanks the SERC for the financial support of his Senior Research Fellowship, and Ian Hayes both the financial support from the Special Studies Program from the University of Queensland and the hospitality of the Department of Computer Science at the University of Manchester, where he visited for the first half of 1993. We would also like to thank the BCS journal *Formal Aspects of Computing* for permission to reuse the NDB material from [Hay92].

References

[AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice Hall International, 1982.

[Bow92] J.P. Bowen. Select Z bibliography. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 367–397. Springer-Verlag, 1992.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[Daw91] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[FJ90] J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 189–210. Springer-Verlag, 1990.

[GLW91] P.H.B. Gardiner, P.J. Lupton, and Jim C.P. Woodcock. A simpler semantics for Z. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 3–11. Springer-Verlag, 1991.

[Hay92] I. J. Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 4(1):76–99, 1992.

[Hay93] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International, second edition, 1993.

[Hoa91] C. A. R. Hoare. Preface. In *[PT91]*, pages vii–x, 1991.

[HW93] I. J. Hayes and L. P. Wildman. Towards libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting, London, December 1992*, Workshops in Computing, pages 37–51. Springer-Verlag, 1993.

[ISO91] ISO. Japan's input on the VDM-SL standardization, April 1991. ISO/IEC JTC1/SC22/WG19-VDM-SL.

[JM93] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. Logic Group Preprint Series 89, Utrecht University, Department of Philosophy, April 1993.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[Lar93] Peter Gorm Larsen. VDM as a mature formal method. Technical report, Institute of Applied Computer Science, April 1993.

[LBC90] J. T. Latham, V. J. Bush, and I. D. Cottam. *The Programming Process: An Introduction Using VDM and Pascal*. Addison-Wesley, 1990.

[MS84] C.C. Morgan and B.A. Sufrin. Specification of the UNIX file system. *IEEE Trans. on Software Engineering*, SE-10(2): 128–142, March 1984. An updated version is available as [Hay93, Chapter 4].

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International, 1991.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991, Vol.2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Spi88] J.M. Spivey. *Understanding Z—A Specification Language and its Formal Semantics*. Cambridge Tracts in Computer Science 3. Cambridge University Press, 1988.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.

[Suf88] B. A. Sufrin. Notes for a Z handbook: Part 1 – the mathematical language, 1988. Programming Research Group, Oxford University.

[Wal90] A. Walshe. NDB. In *[JS90]*, chapter 2, pages 11–46. Prentice Hall International, 1990.

[War93] N. Ward. Adding specification constructors to the refinement calculus. In J.C.P. Woodcock and P.G. Larsen, editors, *Proceedings, Formal Methods Europe'93*, volume 670 of *Lecture Notes in Computer Science*, pages 652–670. Springer Verlag, 1993.

[WB92] J.C.P. Woodcock and S.M. Brien. W: A logic for Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.

[Wel82] A. Welsh. The specification, design and implementation of NDB. Master's thesis, University of Manchester, 1982.

[WH93] M. Woodman and B. Heal. *Introduction to VDM*. McGraw-Hill, 1993.

[Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

A VDM specification

This specification has been adapted from the NDB specification in [FJ90]. In some minor respects (e.g. optional relation names), it is more restrictive than the original [Wel82] (to which the reader is referred for a description of the operations – such as $ADDTUP$ – which are not discussed above).

module NDB

parameters

types
$Value, Esetnm, Rnm : Triv$

exports

operations $ADDES, ADDENT, ADDREL, ADDTUP, DELES, DELENT, DELREL, DELTUP$

definitions

types

$Eid =$ token;

$Maptp = \textsc{OneOne} \mid \textsc{OneMany} \mid \textsc{ManyOne} \mid \textsc{ManyMany};$

$Tuple :: fv : Eid$
$\qquad tv : Eid$;

$Relation = Tuple$-set;

$Rinf :: tp : Maptp$
$\qquad r : Relation$

inv $(mk\text{-}Rinf(tp, r)) \triangleq arity\text{-}match(tp, r);$

$Rkey :: nm : Rnm$
$\qquad fs : Esetnm$
$\qquad ts : Esetnm$

state Ndb of
$esm : Esetnm \xrightarrow{m} Eid$-set
$em : Eid \xrightarrow{m} Value$
$rm : Rkey \xrightarrow{m} Rinf$

inv $(mk\text{-}Ndb(esm, em, rm)) \triangleq$
$\quad \text{dom } em = \bigcup \text{rng } esm \wedge$
$\quad \forall rk \in \text{dom } rm \cdot \{rk.fs, rk.ts\} \subseteq \text{dom } esm \wedge$
$\qquad\qquad \forall mk\text{-}Tuple(fv, tv) \in rm(rk).r \cdot fv \in esm(fs) \wedge tv \in esm(ts)$

init $(ndb) \triangleq ndb = mk\text{-}Ndb(\{\mapsto\}, \{\mapsto\}, \{\mapsto\})$
end

functions

$arity\text{-}match\,(tp, r) \triangleq$
$\quad (tp = \textsc{OneMany} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.tv = t_2.tv \;\Rightarrow\; t_1.fv = t_2.fv) \wedge$
$\quad (tp = \textsc{ManyOne} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \;\Rightarrow\; t_1.tv = t_2.tv) \wedge$
$\quad (tp = \textsc{OneOne} \;\Rightarrow\; \forall t_1, t_2 \in r \cdot t_1.fv = t_2.fv \;\Leftrightarrow\; t_1.tv = t_2.tv)$

operations

$ADDES\,(es : Esetnm)$
ext wr $esm : Esetnm \xrightarrow{m} Eid$-set

```
pre  $es \notin \mathsf{dom}\, esm$
post $esm = \overleftarrow{esm} \cup \{es \mapsto \{\,\}\}$ ;

$DELES\,(es : Esetnm)$
ext wr $esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
    rd $rm : Rkey \xrightarrow{m} Rinf$
pre  $es \in \mathsf{dom}\, esm \wedge esm(es) = \{\,\} \wedge$
    $\forall\, rk \in \mathsf{dom}\, rm \cdot es \neq rk.fs \wedge es \neq rk.ts$
post $esm = \{es\} ⩤ \overleftarrow{esm}$ ;

$ADDENT\,(memb : Esetnm\text{-}\mathsf{set}, val : Value)\; eid : Eid$
ext wr $esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
    wr $em : Eid \xrightarrow{m} Value$
pre  $memb \subseteq \mathsf{dom}\, esm$
post $eid \notin \mathsf{dom}\, \overleftarrow{em} \wedge$
       $em = \overleftarrow{em} \cup \{eid \mapsto val\} \wedge$
       $esm = \overleftarrow{esm} \dagger \{es \mapsto \overleftarrow{esm}(es) \cup \{eid\} \mid es \in memb\}$ ;

$DELENT\,(eid : Eid)$
ext wr $esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
    wr $em : Eid \xrightarrow{m} Value$
    rd $rm : Rkey \xrightarrow{m} Rinf$
pre  $eid \in \mathsf{dom}\, em \wedge$
    $\forall\, t \in \bigcup \{ri.r \mid ri \in \mathsf{rng}\ rm\} \cdot t.fv \neq eid \wedge t.tv \neq eid$
post $esm = \{es \mapsto \overleftarrow{esm}(es) \setminus \{eid\} \mid es \in \mathsf{dom}\, \overleftarrow{esm}\} \wedge$
       $em = \{eid\} ⩤ \overleftarrow{em}$ ;

$ADDREL\,(rk : Rkey, tp : Maptp)$
ext rd $esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
    wr $rm : Rkey \xrightarrow{m} Rinf$
pre  $\{rk.fs, rk.ts\} \subseteq \mathsf{dom}\, esm \wedge$
    $rk \notin \mathsf{dom}\, rm$
post $rm = \overleftarrow{rm} \cup \{rk \mapsto mk\text{-}Rinf(tp, \{\,\})\}$ ;

$DELREL\,(rk : Rkey)$
ext wr $rm : Rkey \xrightarrow{m} Rinf$
pre  $rk \in \mathsf{dom}\, rm \wedge r(rm(rk)) = \{\,\}$
post $rm = \{rk\} ⩤ \overleftarrow{rm}$ ;

$ADDTUP\,(fval, tval : Eid, rk : Rkey)$
ext wr $rm : Rkey \xrightarrow{m} Rinf$
    rd $esm : Esetnm \xrightarrow{m} Eid\text{-}\mathsf{set}$
pre  $rk \in \mathsf{dom}\, rm \wedge$
    let $mk\text{-}Rkey(nm, fs, ts) = rk$ in
    let $mk\text{-}Rinf(tp, r) = rm(rk)$ in
    $fval \in esm(fs) \wedge tval \in esm(ts) \wedge$
    $arity\text{-}match(tp, r \cup mk\text{-}Tuple(fval, tval))$
post let $tup = mk\text{-}Tuple(fval, tval)$ in
       let $ri = \mu\,(\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) \cup \{tup\})$ in
       $rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$ ;
```

$DELTUP\ (fval, tval : Eid, rk : Rkey)$
ext wr $rm : Rkey \xrightarrow{m} Rinf$
pre $rk \in \mathsf{dom}\, rm$
post let $tup = mk\text{-}Tuple(fval, tval)$ in
    let $ri = \mu\ (\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) \setminus \{tup\})$ in
    $rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$

end NDB

B Z specification

This specification has been adapted from the specification of NDB given in [Hay92].

B.1 Entities and entity sets (or types)

$[Eid, Esetnm, Value]$

$Entities$
$esm : Esetnm \mathrel{+\!\!\!\!\!\!\twoheadrightarrow} (\mathbb{F}\ Eid)$
$em : Eid \mathrel{+\!\!\!\!\!\!\twoheadrightarrow} Value$
$\mathrm{dom}\ em = \bigcup(\mathrm{ran}\ esm)$

$\Delta Entities \mathrel{\widehat=} Entities \wedge Entities'$

$\Xi Entities \mathrel{\widehat=} [\Delta Entities \mid \theta Entities' = \theta Entities]$

$ADDES0$
$\Delta Entities$
$es? : Esetnm$
$es? \notin \mathrm{dom}\ esm\ \wedge$
$esm' = esm \cup \{es? \mapsto \{\}\} \wedge em' = em$

$DELES0$
$\Delta Entities$
$es? : Esetnm$
$es? \in \mathrm{dom}\ esm \wedge esm(es?) = \{\} \wedge$
$esm' = \{es?\} \mathbin{⩤} esm \wedge em' = em$

$ADDENT0$
$\Delta Entities$
$memb? : \mathbb{F}\ Esetnm$
$val? : Value$
$eid! : Eid$
$memb? \subseteq \mathrm{dom}\ esm\ \wedge$
$eid! \notin \mathrm{dom}\ em\ \wedge$
$em' = em \cup \{eid! \mapsto val?\} \wedge$
$esm' = esm \oplus \{es : memb? \bullet es \mapsto esm(es) \cup \{eid!\}\}$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DELENT0}} \\ \hline
\Delta \textit{Entities} \\
\textit{eid}? : \textit{Eid} \\ \hline
\textit{eid}? \in \operatorname{dom} \textit{em} \land \\
\textit{em}' = \{\textit{eid}?\} \ndres \textit{em} \land \\
\textit{esm}' = \{\textit{es} : \operatorname{dom} \textit{esm} \bullet \textit{es} \mapsto \textit{esm}(\textit{es}) \setminus \{\textit{eid}?\}\} \\ \hline
\end{array}$$

B.2 A single relation

$$\begin{array}{l}
\textit{Tuple} == \textit{Eid} \times \textit{Eid} \\
\textit{Relation} == \textit{Eid} \leftrightarrow \textit{Eid}
\end{array}$$

$$\textit{Maptp} ::= \textit{OneOne} \mid \textit{OneMany} \mid \textit{ManyOne} \mid \textit{ManyMany}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rinf}} \\ \hline
\textit{tp} : \textit{Maptp} \\
r : \textit{Relation} \\ \hline
(\textit{tp} = \textit{OneOne} \Rightarrow r \in \textit{Eid} \pinj \textit{Eid}) \land \\
(\textit{tp} = \textit{ManyOne} \Rightarrow r \in \textit{Eid} \pfun \textit{Eid}) \land \\
(\textit{tp} = \textit{OneMany} \Rightarrow r^{\sim} \in \textit{Eid} \pfun \textit{Eid}) \\ \hline
\end{array}$$

$$\Delta \textit{Rinf} \mathrel{\widehat{=}} [\textit{Rinf}; \textit{Rinf}' \mid \textit{tp}' = \textit{tp}]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ADDTUPLE0}} \\ \hline
\Delta \textit{Rinf} \\
t? : \textit{Tuple} \\ \hline
r' = r \cup \{t?\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DELTUPLE0}} \\ \hline
\Delta \textit{Rinf} \\
t? : \textit{Tuple} \\ \hline
r' = r \setminus \{t?\} \\ \hline
\end{array}$$

B.3 Multiple relations

$$[\textit{Rnm}]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Rkey}} \\ \hline
\textit{nm} : \textit{Rnm} \\
\textit{fs}, \textit{ts} : \textit{Esetnm} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ndb}} \\ \hline
\textit{Entities} \\
\textit{rm} : \textit{Rkey} \ffun \textit{Rinf} \\ \hline
\forall \textit{rk} : \operatorname{dom} \textit{rm} \bullet \\
\qquad \{\textit{rk}.\textit{fs}, \textit{rk}.\textit{ts}\} \subseteq \operatorname{dom} \textit{esm} \land \\
\qquad (\textit{rm}\ \textit{rk}).r \in \textit{esm}(\textit{rk}.\textit{fs}) \leftrightarrow \textit{esm}(\textit{rk}.\textit{ts}) \\ \hline
\end{array}$$

$\Delta Ndb \mathrel{\widehat{=}} Ndb \land Ndb'$
$\Delta REL \mathrel{\widehat{=}} \Delta Ndb \land \Xi Entities$

$$\begin{array}{|l}
\textit{ADDREL} \\
\hline
\Delta REL \\
tp? : Maptp \\
rk? : Rkey \\
\hline
rk? \notin \operatorname{dom} rm \land \\
\{rk?.fs, rk?.ts\} \subseteq \operatorname{dom} esm \land \\
rm' = rm \cup \{rk? \mapsto (\mu\, Rinf \mid r = \{\} \land tp = tp?)\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{DELREL} \\
\hline
\Delta REL \\
rk? : Rkey \\
\hline
rk? \in \operatorname{dom} rm \land (rm\ rk?).r = \{\} \land \\
rm' = \{rk?\} \mathbin{\lhd\!\!\!-} rm \\
\hline
\end{array}$$

B.4 Promotion of operations

$\Xi RM \mathrel{\widehat{=}} [\Delta Ndb \mid rm' = rm]$
$ADDES \mathrel{\widehat{=}} ADDES0 \land \Xi RM$
$DELES \mathrel{\widehat{=}} DELES0 \land \Xi RM$
$ADDENT \mathrel{\widehat{=}} ADDENT0 \land \Xi RM$
$DELENT \mathrel{\widehat{=}} DELENT0 \land \Xi RM$

$$\begin{array}{|l}
\textit{Promote} \\
\hline
\Delta REL \\
rk? : Rkey \\
\Delta Rinf \\
\hline
rk? \in \operatorname{dom} rm \land \\
\theta Rinf = rm(rk?) \land \\
rm' = rm \oplus \{rk? \mapsto \theta Rinf'\} \\
\hline
\end{array}$$

$ADDTUPLE \mathrel{\widehat{=}}$
$\quad (\exists\, \Delta Rinf \bullet ADDTUPLE0 \land Promote)$

$DELTUPLE \mathrel{\widehat{=}}$
$\quad (\exists\, \Delta Rinf \bullet DELTUPLE0 \land Promote)$

BCS-FACS Christmas Meeting 2006

TEACHING FORMAL METHODS: PRACTICE AND EXPERIENCE

A One Day Workshop at BCS London Offices, First Floor, The Davidson Building, 5 Southampton Street, London WC2E 7HA

FRIDAY 15 DECEMBER 2006

Organised by Oxford Brookes University (OBU) and BCS-FACS

http://www.bcs-facs.org/events/xmas2006.html daduce@brookes.ac.uk

The workshop will give teachers of formal methods an opportunity to discuss their experiences in this area, to share successes and failures, to identify issues in teaching formal methods and discuss how they might be addressed. Topics to be covered include:

+ how to motivate the study of formal methods;
+ techniques for teaching formal methods;
+ handling students with limited mathematics backgrounds;
+ linking formal methods and software development;
+ tools for teaching formal methods;
+ how to assess formal methods.

Papers (4-6 sides A4) are invited outlining experience from which others could benefit; a selection will be chosen by the Programme Committee for presentation (ca. 20 mins) and inclusion in the proceedings to be published in the BCS Electronic Workshops in Computing (eWiC) series after the event. Paper copies of the proceedings will be made available to delegates. Papers should be prepared according to the eWiC guidelines. Style files are available for Latex, Word and HTML. Authors of accepted papers will be required to sign a licence to publish. All these documents are available at http://ewic.bcs.org/organisers/index.htm.

Papers should be submitted as PDF files and sent by email to daduce@brookes.ac.uk.

Organisations and vendors wishing to display literature or demonstrate tools and products are asked to contact the organisers (email: daduce@brookes.ac.uk)

DEADLINES

20 October: Submission of papers; send by email to daduce@brookes.ac.uk, in PDF format
17 November: Notification of selection for presentation/inclusion in proceedings
24 November: Final versions of papers for proceedings
6 December: Registration closes
15 December: Workshop [registration to 10.30 am, close approx. 5.00 pm]

For further information, please see

http://www.bcs-facs.org/events/xmas2006.html

Conference Announcements

The following are sponsored by BCS-FACS and/or considered of special interest to BCS-FACS members:

September 2006

CPA 2006 – 7th Annual Conference on Communicating Process Architecturess
17–20 September
Edinburgh, Scotland
http://www.wotug.org/cpa2006/

October 2006

AWCVS 2006 – 1st Asian Working Conference on Verified Software
29–31 October
Macao, China
http://www.iist.unu.edu/www/workshop/AWCVS2006/

ICFEM 2006 – 8th International Conference on Formal Engineering Methods
30 October – 3 November
Macao, China
http://www.iist.unu.edu/icfem06

FMIS 2006 – 1st International Workshop on Formal Methods for Interactive Systems
31 October
Macao, China
http://fmis.iist.unu.edu/

November 2006

First Alloy Workshop, Colocated with Foundations of Software Engineering
6 November
Submission: 11 September
Portland, Oregon
http://alloy.mit.edu/workshop/

WWV 06 – 2nd International Workshop on Automated Specification and Verification of Web Systems
15–16 November
Cyprus
http://www.dsic.upv.es/workshops/wwv06/

November 2006

ICTAC 2006 – 3rd International Colloquium on Theoretical Aspects of Computing
20–24 November
Gammart/Tunis, Tunisa
http://www.iist.unu.edu/ICTAC2006

December 2006

BCS-FACS Christmas Meeting: Teaching Formal Methods
15 December
Submission: 20 October
London, UK
http://www.bcs-facs.org/events/xmas2006.html

January 2007

POPL 2007 – 34th Annual Symposium on Principles of Programming Languages
17–19 January
Nice, France
http://www.cs.ucsd.edu/popl/07/

February 2007

TAP 2007 – International Conference on Tests and Proof
12–14 February
Submission: 30 September
Zurich, Switzerland
http://tap.ethz.ch/

April 2007

BCTCS 2007 – 23rd British Colloquium for Theoretical Computer Science
2–5 April
Oxford
http://cms.brookes.ac.uk/bctcs2007/

August 2007

CALCO 2007 – 2nd Conference on Algebra and Co-algebra in Computer Science
Submission: 28 January (abstract), 7 February (paper)
20–24 August
Bergen, Norway
http://www.ii.uib.no/calco07/

For further conference announcements, please visit the Formal Methods Europe **website [*http://www.fmeurope.org*], the** EATCS **website [*http://www.eatcs.org*] and the** Virtual Library Formal Methods **website [*http://vl.fmnet.info/meetings*].**

PhD Abstracts

Name	Phan Cong Vinh
Title	Formal Aspects of Dynamic Reconfigurability in Reconfigurable Computing Systems
Supervisor	Prof. Jonathan Bowen & Dr. Ali Abdallah
Institute	Institute for Computing Research, Faculty of BCIM, London South Bank University
Examiners	Dr. Paul Curzon & Prof. Nimal Nissanke
Awarded	27 February 2006
URL	http://myweb.lsbu.ac.uk/~phanvc/papers/phdthesis.pdf
Keywords	Reconfigurable computing, Formal methods

Reconfigurable computing aims to provide support with technological and economic benefits in the design of complex computing systems. Reconfigurable computing can significantly speed up the execution time of dissimilar applications with high flexibility. Reconfigurability of computing provides an extra design dimension to narrow the gap between hardware programming and software programming.

In the last decade, several research attempts have proven that with the appearance of *Field-Programmable Gate Arrays* (FPGA), the reconfigurable computing approach has emerged as a useful option to the traditional approaches based on von Neumann processors or *Application-Specific Integrated Circuits* (ASICs). We consider a rigorous approach to the *dynamic reconfigurability* of *Dynamically Programmable Field Array* (DPGA)-based computing and examine formal aspects of this computing field. More explicitly, our contribution involves the following developments:

- An algebraic structure of a DPGA-based configuration, which is generated by a collection of constructors, its reconfiguration and its reconfiguration mechanism are considered. In other words, we develop an algebra of reconfiguration that models the transformations such as either rotation, reflection or translation in a DPGA.

- The provable algorithms allow one to partially change a DPGA-based configuration at runtime without stopping the operation of the full system.

- A unifying framework of reconfiguration is seen as a coalgebraic model. In which the reconfiguration processes are observed using the concept of functional stream derivatives.

- One of the combinational features of reconfiguration touches on the regularly repeated arrangements of the transformations satisfying specified constraints. The stream calculus and coinduction are used to solve the problems regarding arrangements of transformations in an expressive, powerful and uniform way.

- Regarding the flowware in configware engineering, the notion of behavioural function in stream calculus is used to support the semantics for behaviours of the computing based on data flow at the Register Transfer Level (RTL). Moreover, the coinductive proof principle is used to compare such the behavioural functions and a coinductive approach to verifying flowware synthesis is introduced.

In addition, we recognise that high complexity to formally verify the entirety of a reconfigurable computing process is one of the barriers. Hence,

- In another aspect, using Partial Order-based Model (POM) to support the semantics for the Register Transfer Level (RTL) is also developed and through this semantics we propose a validation method to test the correctness of an RTL synthesis result.

FACS Evening Seminars 2006

4 September
The Computer Ate my Vote
Prof. Peter Ryan, University of Newcastle

6pm Start

9 November
Prof. Ursula Martin, Queen Mary University of London
5.45pm Start

Seminars are held at BCS London Offices, 5 Southampton Street, London WC2E 7HA.

Dates for your diary in 2007:

7 February 2007 – seminar by Michael Jackson
21 March 2007 – seminar by Egon Börger

Further dates to be announced soon. Please see http://www.bcs-facs.org/events/EveningSeminars for up-to-date information on the seminars.

FACS membership application/renewal (2006)

Title (Prof/Dr/Mr/Ms) _____ **First name** _____________ **Last name**____________

Email address (required for options * below)________________________________

BCS membership No. (or sister society name + membership number)

____

Address ____

____

____

Postcode ______________ **Country** ____________________________

I would like to take out **membership to FACS** at the following rate:

- ❑ £15 (Previous member of BCS-FACS now retired, unwaged or a student)
- ❑ £15 (Member of BCS or sister society with web/email access)*
- ❑ £30 (Non-member or member of BCS or sister society without web/email access)

ALL MEMBERS WILL RECEIVE FREE ELECTRONIC ACCESS TO THE *FORMAL ASPECTS OF COMPUTING* JOURNAL UNTIL THE END OF DECEMBER 2006

I would like to subscribe to **Volume 18 of the FAC journal** (paper copy) at the following rate:

- ❑ £48

The total amount payable to BCS-FACS in **pounds sterling** is **£ 15 / 30 / 63 / 78** (delete as appropriate). I am paying by:

- ❑ Cheque made payable to **BCS-FACS** (**in pounds sterling**)
- ❑ Credit card via PayPal (instructions can be found on the BCS-FACS website)
- ❑ Direct transfer (in **pounds sterling**) to:

Bank: Lloyds TSB Bank, Langham Place, London
Sort Code: 30-94-87
Account Number: 00173977
Title of Account: BCS-FACS

If a receipt is required, please tick here ❑ and **enclose** a stamped self-addressed envelope.

Please send completed forms to:

Dr Paul P Boca
PO BOX 32173
LONDON N4 4YP
UK

For FACS use only

Received by FACS	Date:	Initials:
Sent to Springer	Date:	Initials:
Actioned by Springer	Date:	Initials:

FACS Committee

Jonathan Bowen
FACS Chair
ZUG Liaison

Jawed Siddiqi
Treasurer

Paul Boca
Secretary and
Newsletter Editor

Executive Officers

Roger Carsley
Minutes
Secretary

John Cooke
FAC Journal
Liaison

John Fitzgerald
FME Liaison
SCSC Liaison

Judith Carlton
Industrial Liaison

Margaret West
BCS Liaison

Rick Thomas
LMS Liaison

Mark D'Inverno
Model-based
specification

John Derrick
Refinement

Rob Hierons
Formal methods
and testing

FACS is always interested to hear from its members and keen to recruit additional Committee members. Presently we have vacancies for officers to handle publicity and help with fund raising, and to liaise with other specialist groups such as the Requirements Engineering group and the European Association for Theoretical Computer Science (EATCS). If you are interested in helping the Committee, 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
Faculty of BCIM
Borough Road
London SE1 0AA
United Kingdom

T +44 (0)20 7815 7462
F +44 (0)20 7815 7793
E info@bcs-facs.org.uk
W www.bcs-facs.org

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

Please feel free to discuss any ideas you have for FACS or voice any opinions openly on the FACS mailing list [FACS@jiscmail.ac.uk]. You can also use this list to pose questions and to make contact with other members working in your area. Note: only FACS members can post to the list; archives are accessible to everyone at http://www.jiscmail.ac.uk/lists/facs.html .

Coming Soon in FACS FACTS….

Articles

Conference Announcements

Conference Reports

Book Reviews

Details of upcoming FACS Evening Seminars

Book Announcements

Return of FX Reid

And More…

Back to Contents page