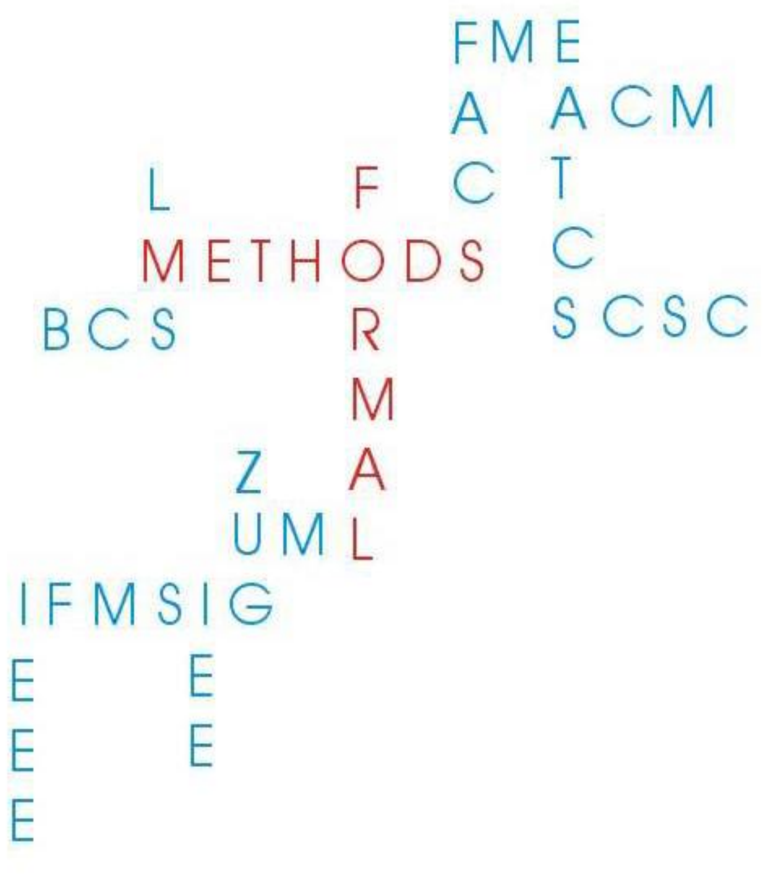
Issue 2017-1

August 2017

FACS FACTS



Formal Aspects of Computing Science Specialist Group

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

ISSN 0950-1231

About FACS FACTS

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

Submissions to FACS FACTS are always welcome. Please visit the newsletter area of the BCS FACS website for further details at:
http://www.bcs.org/category/12461

Back issues of FACS FACTS are available for download from:
http://www.bcs.org/content/conWebDoc/33135

The FACS FACTS Team

Newsletter Editors

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

Editorial Team

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

Contributors to this issue

Paul Boca, Jonathan Bowen, Tim Denvir, Cliff Jones, The Computer Laboratory, University of Cambridge, Brian Monahan, Colin Snook, Margaret West.

BCS-FACS websites

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

If you have any questions about BCS-FACS, please send these to Paul Boca <paul.boca@googlemail.com>

Editorial

A warm welcome to our first issue of FACS FACTS for 2017! It seems that one can hardly turn on the TV or peruse the Internet without hearing about a fresh advance in advanced computing or artificial intelligence these days – and this issue appears to be no exception to that trend.

Our first article is a timely and extensive review from Prof. Cliff Jones (to whom we are extremely grateful) of the recently published book The Turing Guide. As I'm sure we all appreciate, Alan Turing was a pioneer in so many aspects of what has become Computing Science, not least including Artificial Intelligence, of course. Turing's famous 1950 paper *Computing Machinery and Intelligence*, introducing the well-known 'Turing Test', was an early landmark of what was then called Machine Intelligence and which later became better known as Artificial Intelligence. This subject has of course seen a tremendous resurgence recently, primarily through the Machine Learning approach which deploys compute-intensive statistical optimisation techniques that can extract and generalise patterns from vast quantities of (typically labelled) data.

Our second contribution is a report by Margaret West on a (rare) talk given at Royal Holloway by another pioneer of Artificial Intelligence, Prof. Robert Kowalski. This talk concerns the fascinating contrast between Computational Logic and Human Thinking, a topic which Kowalski has extensively written on over many years.

One of your editors, Tim Denvir, gave an insightful and well-attended talk at the British Computer Society offices in London in May this year, concerning his extensive experience of applying Formal Methods in industry within a wide number of contexts. An interesting point from Tim's talk is the observation that issues involving concurrency and parallelism continue to provide a substantial and enduring challenge to software development methodology. Our third contribution in this issue is an extended essay by Tim himself, based upon that talk.

Our next contribution is a report by Jonathan Bowen on a talk in May by Prof. Dr. Reiner Hähnle, of TU Darmstadt, Germany concerning The KeY Formal

Verification tool. This provides extensive automated support for deductive verification for sequential Java and is based upon a rich program logic for Java source code. Interestingly, the tool featured prominently in finding subtle bugs in some very widely used library code (e.g. the Timsort algorithm, a sorting algorithm optimized for partially sorted arrays provided in various well-known libraries).

In a similar vein, Colin Snook of the University of Southampton presented a talk at the BCS concerning a tool called iUML-B that provides advanced integrated support for constructing formal specifications for Event-B within the agile notation, UML.

It is with great sadness that we recently learnt of the death on 22nd August of Professor Emeritus Michael Gordon, FRS, of the University of Cambridge, following a short illness. Mike was a much respected teacher and researcher, having led the research in Cambridge that gave us the HOL system which defined the template for many future interactive theorem proving systems, as well as leading its application to machine hardware and systems design correctness proofs. We have reproduced his obituary on page 14, with kind permission of the Computer Laboratory, University of Cambridge.

Finally, BCS-FACS is hoping to support ABZ 2018 and FM 2018 - details of these events can be found here:

ABZ 2018: https://www.southampton.ac.uk/abz2018/

FM 2018: http://www.fmeurope.org/?p=613

Most FACS seminars take place in the offices of the BCS in the Davidson Building, Southampton Street. These excellent facilities are conveniently situated in Central London close to Covent Garden and we would like to thank the BCS for making these available to us. We look forward to seeing you there!

Brian Monahan, Co-editor

Review of *The Turing Guide*, Jack Copeland, Jonathan Bowen, Mark Sprevak, Robin Wilson and others, Oxford Press, 2017.

This is an interesting book which I am glad to have read. Unsurprisingly, in a volume that contains 42 chapters by a large group of authors, some portions of the book will -for any reader- be more interesting and informative than others. I should also add that I think this book would be best served by dipping into chapters that appeal to the reader. As a reviewer, I have read the whole book in chapter order in a relatively short period of time.

Let me start with the pre-conditions for readers who might find this book enjoyable and perhaps informative. Because the chapters in this collection are written independently, there is some overlap, even some contradiction or differences in assessment. Particularly if the reader is dipping into the chapters, this will be less aggravating than in a cover-to-cover read. More worrying is the difficulty of creating a full picture from just this collection. I had read Hodges' excellent biography of Alan Turing [1] and suggest that anyone who has not done so would be well advised to make this preparation because they will then benefit more from reading *The Turing Guide.*

There is excellent material on the various aspects of Alan Turing's wide range of contributions:

- His seminal work on the *Entscheidungsproblem* and what we now know as *Turing machines*
- Wartime work on cryptography and cryptanalysis
- Efforts on the design of physical computers
- Early thoughts on what we now call "Artificial Intelligence" (AI)
- Ahead-of-his-time ideas on biological growth
- Work in mathematics itself

A word of warning is in order about the material on Turing's mathematical contribution – this is probably the hardest section for a non-expert to understand.

For me, much the most interesting material comes from those who in one way or another were close to Turing. For example, Chapter 2 (*The man with terrible trousers*) by Alan Turing's nephew Sir John Dermot Turing speaks frankly about Alan's estrangement from the family and poignantly about the understandable desire to minimise the distress to Alan's mother of both his "criminal" prosecution and the potential scandal at the time of Alan's death. In all the debate about whether the death was or was not suicide (or something more sinister), I at least had never fully appreciated the family dimension of the last years of Alan's life. Sir John writes as a lawyer and points out that Alan's death came just before the Wolfenden Committee began the process of overturning the inhuman laws relating to homosexuality (that I still find hard to believe subjected any person to the treatment meted out in my lifetime to Alan and many others).

On a more cheerful note, memories from Peter Hilton (*Meeting a genius*), Eleanor Ireland (*We were the world's first computer operators*) and Jerry Roberts (*The Testery: breaking Hitler's most secret code*) also brought this reader closer to Alan Turing as a living person in a way that complements Hodges' biography.

I also welcome Hilton's firm "Alan Turing was the acknowledged leading light of the [Enigma code] team. However, I must emphasise that we were a team – this was no one-man show". This is a (and by no means the only) correction to the journalistic tendency to describe Turing as "The man who invented x" (for too many instances of "x").

In addition to the personal contacts, some chapters I found particularly rewarding are:

- Doron Swade's (*Turing, Lovelace and Babbage*) is a very clear account – in wonderfully lucid prose – of a large historical perspective of computing machines.

- Brian Randell’s chapters 8 and 17; for anyone who has not heard his account of uncovering the *Colossus* story, these chapters are strongly recommended.
- The material on *Biological Growth* (Margaret Boden’s *Pioneer of artificial life* and the following chapter *Turing’s theory of morphogenesis* by Wooley, Baker and Maini) are very clear expositions of material outside my normal reading and from which I learned a great deal.
- Ivor Grattan-Guinness’ *Turing’s mentor: Max Newman* is a (short) contribution that is excellent value.

No sane person would deny that Turning was a genius. However, the desire of readers to find heroes who change the world single-handedly and the temptation of more journalistic writers to attribute progress to the contributions of single individuals serves neither historical accuracy nor, ultimately, the reputation of the individuals. There have been, for example, several post-Turing-centenary articles that have pushed back against the view of Turing as *the* father of the subject now known as Computer (or Computing) Science.

As I made clear at the outset, I recommend the *Turing Guide*: it has a lot of interesting material even if it is not uniformly well argued. Having made that positive evaluation, I allow myself a few minor reservations:

- There are some odd splits of material that don’t serve to help the reader’s understanding e.g. Randell’s material (Chaps 8 and 17); Proudfoot’s (Chaps 28 and 30); Simpson’s (Chaps 13 and 38).
- As someone who worked in Manchester University for 15 years and who discussed the early history with Tom Kilburn in the run-up to the 1998 anniversary of the “Baby”, I could wish that it were possible to form a panel of those involved to balance some of the statements made about Manchester machines.
- To my eyes, the typesetting of quotations (with, for example, no inset) makes them difficult to read.

In addition to the Manchester "disclaimer" I should state that: I am a colleague and friend of Brian Randell and a contributor to yet another book related to Alan Turing [2].

Cliff B. Jones
2017-04-19

[1] *Alan Turing: the Enigma* A. Hodges, Burnett, 1983 and Simon and Schuster, 1988

[2] *Alan Turing: His Work and Impact* edited by S. Barry Cooper and Jan van Leeuwen, Elsevier, 2013

Logic, Artificial Intelligence and Human Thinking

Royal Holloway Distinguished Seminar

Professor R. A. Kowalski

February 13th 2017

Abstract Symbolic logic has been used in artificial intelligence over the past 60 years or so, in the attempt to program computers to display human levels of intelligence. As a result, new forms of computational logic have been developed, which are both more powerful and more practical. The new computational logic is the logic of an intelligent agent whose mission in life is to make its goals true, by performing actions to change the world, in the context of changes in the world that are outside its control. For this purpose, the agent uses its beliefs in logical form both to reason forwards, synthetically to derive consequences of its observations and candidate actions, and to reason backwards, analytically to reduce goals to subgoals, including actions.

I will argue that computational logic can be used not only for artificial intelligence, but for more conventional computing; and because it improves upon traditional logic, it can also be used for the original purpose of logic, to help people improve their own natural intelligence.

Biography

Robert Kowalski is Emeritus Professor and Distinguished Research Fellow at Imperial College London. He studied at the University of Chicago, the University of Bridgeport, Stanford University, the University of Warsaw, and the University of Edinburgh, where he completed his PhD in 1970. He joined Imperial College in 1975, becoming Professor Emeritus in 1999.

During the 1980s, Kowalski was heavily involved in the British response to the Japanese Fifth Generation Project. He also served as an advisor to the UNDP Knowledge Based Systems Project in India, and to DFKI, the German Institute for Artificial Intelligence. He co-ordinated the European Community Basic Research Project, Compulog, and was the founder of the European Compulog Network of Excellence. More recently he has been an advisor to the Department of Immunization, Vaccines and Biologicals, of the World Health Organization in Geneva.

Kowalski's early research was in the field of automated theorem-proving, leading to the development of logic programming in the early 1970s. His later research has focused on the use of logic programming for knowledge representation and problem solving, including work on the event calculus, legal reasoning, abductive reasoning and argumentation. His current work is aimed at developing a unified, logic-based framework for artificial intelligence, databases and programming. The philosophical background for this work is presented in his 2011 book *Computational Logic and Human Thinking – How to be Artificially Intelligent.*

Kowalski is a Fellow of the Association for the Advancement of Artificial Intelligence, the European Co-ordinating Committee for Artificial Intelligence, and the Association for Computing Machinery. He received the IJCAI (International Joint Conference of Artificial Intelligence) award for Research Excellence in 2011, and the Japanese Society for the Promotion of Science Award for Eminent Scientists for 2012-2014.

Talk

The subject of the talk was a new form of Computational Logic which is capable of capturing human thought processes [1] and the talk commenced with a brief review of many different theories of human (everyday) thought including:

(i) The role of Boole's "The Laws of Thought" in the logic of Sherlock Holmes;

(ii) Intelligent Agents – where all goals are made "true" via the actions of that agent where the given actions have been generated using observations of the outside world and the agent's belief;

(iii) The use of forward and backward reasoning: in Barbara Minto's "The Pyramid Principle" it is recommended that in promoting an idea in writing it is best to present the chosen solution first and *then* supply supporting arguments. This tool is used at McKinsey and is an example of *backward reasoning*.

Kowalski opined that the new form of Computational Logic should constitute **both** Goals (Production Rules) **and** Beliefs (Logic Programs) for achieving a goal the previous state of the system is overwritten and production systems do not have a logical meaning.

Logical Production Systems (LPS – see http://lps.doc.ic.ac.uk/) is a logic based Computer Language developed by Imperial College as part of CLOUT (Computational Logic for Use in Teaching). For explanation – see https://www.doc.ic.ac.uk/~rak/papers/LPS%20with%20CLOUT.pdf

Two kinds of system are combined: Logic based systems and State Transition Systems – the result being the utilisation of computation in generating a model of the world. The model is described by logic programs and reactive rules where LPS combines logic with destructive updates. Logic programs are *declarative* where:

> *If A then B*

means:

> *If A is TRUE then B is TRUE*

whereas State Transition systems are *imperative* and can be described by reactive rules where

> *If A then B*

means a change of state i.e.

> *If A holds then do B.*

LPS unites the above two kinds of system which otherwise have no obvious relationship. State transition systems are common to all areas of computing – an example being Artificial Intelligence where states are "facts" – an agent's belief which can be eliminated or initiated by events where in LPS events and states are time-stamped.

An example was provided to illustrate – modelling the situation where

> *IF you want to go home for the weekend and you have bus fare THEN you can catch a bus.*

The information about the goal *go home for the weekend* is represented in LPS as:

> You go home from T1 to T2
> if you have the bus fare at T1,
> you catch a bus from T1 to T2.

In order to represent human thought in LPS it is necessary to analyse the structure of (for example) instructions of what to do in an emergency situation. An illustrative example was given – a safety notice on the London Underground comprising four sentences.

Emergencies

1 Press the alarm signal button
to alert the driver.

2 The driver will stop
if any part of the train is in a station.

3 If not, the train will continue to the next station,
where help can more easily be given.

4 There is a 50 pound penalty
for improper use.

The purpose of the notice is to regulate passengers' behaviour – in a similar manner to how instructions control a computer program. Thus the notice needs to be clear to a passenger and in a similar manner the logic needs to be understandable to a computer. In order for this to be so the first sentence is written in an imperative manner where backwards reasoning decomposes the problem to a sub-problem (goal-reduction):

you alert the driver
if you press the alarm signal button

Sentences 2 and 3 can be expressed via the two alternatives using the information implicit in sentence 1:

the driver will stop the train in a station
if you alert the driver
and any part of the train is in the station

The driver will stop the train at the next station
and help can be given there better than between stations
if you alert the driver
and not any part of the train is in a station

where forward reasoning is used to derive logical consequences.

Both forward and backward reasoning are used to derive the logic of sentence 4 which becomes

press the alarm signal button improperly
to receive a 50 pound penalty

The speaker concluded with a few remarks about LPS which combines and reconciles declarative and imperative languages. It is a language for programming, databases and knowledge representation and problem solving in AI. It is a practical logical framework for computing which has been used in teaching – CLOUT for example – and while not full-scale can be extended.

During the talk the use of LPS was demonstrated by examples which can use it ... for example dining philosophers, bubble sort, natural language, bank transfers. The LPS demonstration can be found at http://lpsdemo.interprolog.com/

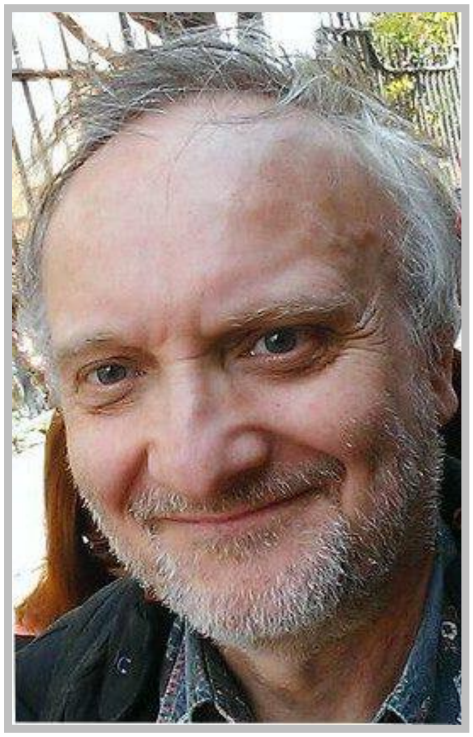
[1] *Computational Logic and Human Thinking – How to be Artificially Intelligent* by Robert Kowalski, Cambridge (2011).

Margaret West
University of Huddersfield

Michael JC Gordon FRS

Professor of Computer Assisted Reasoning
28 February 1948 – 22 August 2017

https://www.cl.cam.ac.uk/misc/obituaries/gordon/



Professor Michael Gordon was a pioneer in the field of interactive theorem proving, with a focus on hardware verification. This field is concerned with certifying system designs by proving their correctness mathematically. Mike Gordon shaped this field from the beginning, demonstrating the feasibility of hardware verification on real-world computer designs. His students extended the work to such diverse areas as the verification of floating-point algorithms, the verification of probabilistic algorithms and the verified translation of source code to (necessarily correct) machine language code. In recognition of his achievements, he was elected to the Royal Society in 1994, and he continued to make valuable contributions until the end of his career.

In the 1970s, as a postdoctoral researcher at Edinburgh University, Mike Gordon was part of the team that built Edinburgh LCF. This was an interactive theorem prover: a program for undertaking formal proofs in a logical calculus (the Logic for Computable Functions). And it was the first of its kind. Although the LCF calculus soon fell out of favour, the architecture of Edinburgh LCF is now almost universally adopted by today's interactive provers. This early project also introduced the ML family of functional programming languages.

Mike met his wife Avra during his first post-doc in 1974, a year spent with John McCarthy at the Stanford Artificial Intelligence Lab where Avra was a Research Assistant. They were colleagues at Edinburgh and Cambridge until Avra retired in 1991 to raise the family.

Mike Gordon was appointed to a Lectureship at Cambridge in 1981. There he turned his attention to hardware, introducing first LCF_LSM (Logic for Sequential Machines) and then HOL (Higher Order Logic). One of his key contributions was to demonstrate the effectiveness of higher order logic as a general formalism for verification, replacing earlier specialised formalisms. At the time, first order logic was preferred both by logicians themselves and by the AI community; Mike demonstrated that higher order logic could be implemented effectively and used to specify hardware designs from the gate level right up to the processor level, as well as abstract hardware specifications. A steady stream of PhD students extended the applicability and power of the HOL system to unimagined levels. Cambridge promoted Mike to Reader in 1988 and Professor in 1996.

The impact of his work, along with that of the students and colleagues, is worldwide. Techniques that originated in his group at Cambridge are used by major chip vendors and have deeply influenced the entire field of interactive theorem proving.

Mike Gordon's colleagues and students will remember him as an attentive and supportive listener, of unfailing kindness and generosity. He is survived by his wife, Avra Cohn, and by their two children Katriel and Reuben Cohn-Gordon.

The Computer Laboratory, University of Cambridge

Fifty Years of Formal Methods in Software Engineering

A Personal View

Tim Denvir

(Photograph by Jonathan Bowen)

This article records the content of a talk given at the BCS on 1 March, 2017. I have based the material on my personal experience and make no claim to present a comprehensive account. Much important work has been done by many people over the period across the world, that I have not been able to mention here. A video of the talk, in two parts, is available here:

Part 1: https://www.youtube.com/watch?v=n3FWRjHmSXU

Part 2: https://www.youtube.com/watch?v=tf8oI8-hUEs

I completed my maths degree in 1962. Jobs for new maths graduates used to be limited to school-teaching and, for the more seriously clever, being an academic mathematician. But in the late fifties and early sixties, computers began to be built on a commercial scale and someone had to write programs for them, and so the software departments of computer manufacturers were full of young maths graduates. There were no computer science degree courses yet, just diplomas, some of them post-graduate, some HNC and so on. So we young mathematicians were very open to the idea of treating programs as mathematical objects.

There were just three main computer science journals then: the ACM Communications and Journal and the BCS Computer Journal. Most of us read just about every paper in all of them. Some fifteen years later, in charge of a software development group at STL, I realised that we had 41 journals on regular order from the firm's library and none of us had time to read any of them. But in the late 50s – early 60s the emphasis of the CS research publications was on formal languages and automata theory. Russian computer scientists were prolific in publishing papers at this time. A frequent author in these journals, always with something original to say, was a young American, Ben Wegbreit. It was the tradition then for an author's photo to appear at the end of a journal paper, and Wegbreit's young bearded face always looked enthusiastic, cheerfully searching. Then suddenly, it seemed, he published no more[1].

Formal languages were defined by grammars, following and inspired by the notations of Chomsky. Born in 1928, Noam Chomsky is a linguist, philosopher, cognitive scientist, social critic and more; in 1952 – 1957 he devised and developed a theory and classification of grammars[2], alongside his theories of learning. The Chomsky-style grammars were ideal for defining the syntax of programming languages and were rapidly taken up by computer scientists. There was a clear relation between finite state automata and formal grammars: given a grammar, one can immediately derive an automaton which generates (or accepts) sentences conforming to that grammar. The syntax definition notation, BNF (Backus Normal/Naur Form), closely followed Chomsky's ideas and was used from 1958 in the definitions of Algol 60[3].

Formal grammars, such as BNF, could be used for driving the construction of the syntax analysis phase of a compiler. Numerous automatic systems for generating a parser for a language based upon its grammar have been constructed, but perhaps the first was the *Compiler-Compiler* by Brooker and

1 A web search suggests he went into a successful commercial career and continues to this day.

2 e.g. Noam Chomsky, *Syntactic Structures*, Mouton & Co., 1957.

3 Preliminary report – International Algebraic Language, Comm. Assoc. Comp. Mach. 1,No. 12 (1958),8; Report on the Algorithmic Language Algol by the ACM Committee on Programming Languages and the GAMM Committee on Programming, ed. A. J. Perlis & K. Samuelson, Numerische Mathematik Bd. 1, S. 41-60 (1959).

Morris implemented on the Manchester Atlas and its siblings in 1963[4].

So far there was no large scale attempt to find a way of defining the semantics of languages, which was perhaps the more crucial issue; one may write a program which is syntactically correct, but if it does not do what one intends, the effort is to no avail and will require much work to rectify.

Analysis of syntax was a much easier problem to grapple with than semantics.

Some of the most canonical work on formal grammars and automata was done by John Hopcroft and Jeffrey Ullman, culminating later in their book *Formal Languages and Their Relation to Automata*[5].

A brief note on the available computer systems and hardware may be useful at this point. Computers were not interactive: the user did not sit at a desk interacting with a computer, instead you would prepare a “job” for the computer to do, and either hand a script of a program to a human operator and ask for it to be compiled and run, or, if there was a more sophisticated system, submit a codified job description for the computer to interpret along with any program script and data. This was called a “Batch Operating System”. Then with luck a few hours later, with less luck the next day, you would receive the results. In 1958 computer electronics built with discrete semiconductors were being commercially built and used, but machines whose electronics were built around valves (thermionic vacuum tubes) were still much in use, notably the Atlas machines, in London, Cambridge and Manchester. These were among the largest and most sophisticated machines in the world, costing some £3-4M, with central processor time priced at £900 per hour (something like £20,000 in today's money). They were never switched off; with thousands of valves, some would always blow on power on or off and would have to be replaced before work could start again. Integrated circuits were being researched in the late 1950s and started to come into production some five to seven years later. The Cambridge Atlas, named Titan, by 1969 did have a form of interactive interface: the user could sit at a terminal and submit a job directly, obtaining the results

4 Brooker, R .A.; MacCallum, I. R.; Morris, D.; Rohl, J. S. (1963), "The compiler-compiler", Annual Review in Automatic Programming, 3: 229-275

5 Hopcroft, John E.; Ullman, Jeffrey D. (1968). *Formal Languages and Their Relation to Automata*. Addison-Wesley.

in about half an hour. This was an absolute wonder and speeded up work by a substantial factor.

The late 50s and early 60s saw the arrival of high-level languages, COBOL (1959), Fortran (1957), Algol 60. During the 60s they came to be used more and more, with other languages being defined as time went on. But even in the mid-seventies, in some sectors such as telecoms and the more engineering-based application areas, machine code and autocodes were still being used for applications. It seemed that some application sectors were reluctant to move forward.

The defining of semantics of languages was slow in coming. The Algol 60 report attempted to state unambiguously what each kind of statement did. It used English to do so, but in a manner that was clearly inspired by Church's λ-calculus[6], that is, it defined a transformation of the script, for example replacing formal parameters in the body of a procedure with their corresponding actual parameters from the procedure call, expanding loops, etc. The first indications of program language semantics that came to the notice of us software engineers were the work of Tony Hoare and Edsger Dijkstra.

Two canonical papers were published in the late 60s: *Goto Statement Considered Harmful* by Dijkstra[7], and *An Axiomatic Basis for Computer Programming*, by Hoare[8]. Dijkstra subsequently followed up his earlier paper with one on *Guarded Commands* in 1975[9]. This introduced the idea of pre- and post-conditions, quite strongly related to the logical components of Hoare's triples in *Axiomatic Basis*, and gave the means of proving sequences of statements, and hence programs, correct with respect to overall pre- and post-conditions. The latter prompted the idea of a formal specification of a program.

I have to admit that when Tony Hoare's paper, *Axiomatic Basis*, was first published in 1969, a number of us reacted by saying, "Was that worth

6 A. Church, "A set of postulates for the foundation of logic" *Ann. of Math. (2)*, 33 (1932) pp. 346-366.

7 E. W. Dijkstra, "Goto Considered Harmful", Letter to the Editor, Communications of the ACM Vol. 11, pp 147-148, 1968.

8 C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", Communications of the ACM Vol. 12, pp 576-580, 1969.

9 E. W. Dijkstra, Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, Communications of the ACM Vol 18, pp 453-457.

publishing? It's all a bit obvious!". But of course it turned out to be ground-breaking. Maybe what separates these intellectual giants from the rest of us is knowing what will be important. But in the mid-seventies a number of us were using Dijkstra's pre- and post-conditions to prove small programs correct. Then, I was drafted on to an ISO committee which was trying to reach agreement about a standard for the CHILL language. CHILL was a high-level language devised for use in telecommunication systems. For too long this sector had been backward, using machine codes and autocodes. There was a general recognition that it was time to move to high-level languages, but there was also a desire to seek some standard, suitable language with real-time features. CHILL had some of the features of Ada, which had been adopted by the US Department of Defense in order to reduce the large number of languages used in embedded defence applications.

I was working for ITT at STL, one of their research laboratories, at the time, and when on the CHILL committee ITT suggested that I attend a Winter School on Abstract Software Specifications in Copenhagen, January 1979. This event was to be a turning point for me and my STL colleague, Bernie Cohen. There was a great line-up of talent at the winter school[10] and it was not surprising that the event had a profound effect. My immediate colleagues and I were particularly impressed with VDM, which was the subject of lectures by Cliff Jones and Dines Bjørner. VDM, the Vienna Development Method[11], was derived from VDL. VDL, the Vienna Definition Language, was developed to define the semantics of PL/I at the IBM Vienna laboratories.

There is a relationship between formal semantics, specification, and proof of program correctness. If you can define the meaning of a language, and thus of a program written in it, you can formulate a specification of what the program is to do. From that a program which meets the specification can be constructed through a process of successive refinement.

The following year, in 1980, a few of us in the software research group at STL persuaded our management that we could hire the services of Cliff Jones to

10 Dines Bjørner, Cliff Jones, Steve Zillies, Joe Stoy, Peter Lucas, Peter Lauer, Barbara Liskov, Gordon Plotkin, Rod Burstall, David Park, O-J Dahl, Peter Mosses, and others.

11 Cliff Jones, *Software Development, a Rigorous Approach*, Prentice-Hall 1980.

help us apply VDM to telecoms projects. From this highly productive consultancy we developed our own courses in VDM and discrete mathematics, gave them internally within STL, then more widely through STC. Most telecoms engineers had degrees in engineering or electronics, a few in computer science, which all entailed considerable mathematics, but more the traditional applied maths rather than set theory, logic etc. We therefore perceived a need for a short course on discrete maths. The production departments in STC received the courses with some enthusiasm. We also experimented with the use of Z in one project, using consultancy from Bernard Sufrin and Carrol Morgan from the Oxford Programming Research Group. STL, the research laboratory of STC, had 1,000 employees, and such was the variety of its operations, its own medical department and company fire brigade. With diverse departments and their external contacts, 30,000 visitors came to STL each year. The visitors' administration clerk kept track of all these visitors and the various facilities provided to them, such as a visitor's lunch or a company car to ferry them to the railway station or airport, and the charging to department budgets for these services. The visitors' clerk did all this using a manual system, card index files and so forth. We asked the PRG if they could begin to convert this to a computer-based system by finding out the requirements from the, entirely non-technical, visitors' clerk, constructing a specification of the system in Z, playing back to the clerk the features of the system they proposed, and overseeing us implement the system in Pascal from the specification in Z. Bernard Sufrin and Carrol Morgan embarked on this task with a will, communicating effectively with the clerk, and discussing the experience with us throughout the exercise.

The telecoms industry had its own design language, SDL, which was the subject of a CCITT[12] standard. Telecoms was in general accepting and in favour of standards, for otherwise electronic communications across borders of all kinds would not be possible. This language, however, was ad hoc, concrete in the sense that a design expressed in it would heavily influence the implementation, and it began to need some kind of “cleaning up”. Robin Milner proposed having a joint enterprise whereby the LFCS at Edinburgh University would supply a researcher seconded to STL in order to apply academic research in an industrial

12 CCITT, the international telecoms standardisation body.

setting, and strengthen experience on both sides of the fence. This was the beginning of a general initiative by both the UK government, in the shape of a joint enterprise between the DTI and the EPSRC, and the European Commission, whose Framework series likewise emphasised collaboration between academia and industry. The idea was that industry would get an intellectual boost from academia, and academia would keep focused on research which would ultimately prove "useful".

Our budget at STL benefited a lot from this arrangement, giving us a qualified researcher at cost, instead of at a loaded rate, a saving of some 67%. I readily agreed. Robin set about recruiting someone for the post straight away, and after a few months came up with the name of Mike Shields. I had met Mike before in 1979 at a conference on the Semantics of Concurrent Computation[13] in Evian, and had been most impressed. Mike came to our group in STL for a two-year spell. We first sent him on an ITT course on "Telecommunications Systems Planning", which most of us in STL had attended. Despite its pragmatic, horny-handed character, Mike was enthusiastic about the course and found it stimulating. After working on a few internal projects, he became involved with SDL and its ongoing definition and development. When I looked up the current standard on SDL years later I was delighted to see that it had changed out of all recognition from those early days. It was no longer ad hoc: with a large measure of formality in its definition, it was more abstract, i.e., less implementation-biased[14]. Mike Shields' intervention played a substantial part in this improvement.

In the 1970s the US Department of Defense had let a substantial and rigorous study to reduce the number of languages used in embedded computing projects. A series of documents, each more specific than its predecessor, were produced elaborating the requirements for the desired language; these requirements were named *Strawman*, *Woodenman*, *Ironman*, and finally in 1978, *Steelman*. They first concluded that no existing language met the Steelman requirements, and so invited proposals to define and ultimately implement a new language. Four contractors were shortlisted to develop their

13 Gilles Kahn, Ed.: "Semantics of Concurrent Computation", Proceedings of the International Symposium, Evian, France, July 2–4, 1979, LNCS 70, Springer 1979.

14 See e.g. https://www.itu.int/rec/T-REC-Z.100/en

proposals, and the corresponding proposed languages were called *Red* (Intermetrics), *Green* (CII Honeywell Bull led by Jean Ichbiah), *Blue* (SofTech), and *Yellow* (SRI International). “Green” won the competition and became the language Ada. In 1981 the DTI let a contract to a consortium of institutions, academic and industrial, to do some substantial work on investigating the use of Ada. At STL our software research department felt a bit miffed that the DTI had not invited us and quite a few other competent contenders to do this work, or even bid for it, and we got together with several other institutions, formed a consortium, which we called *Augusta* after Ada Lovelace's second name, to complain to the DTI. The DTI were embarrassed enough to let an additional, albeit smaller, contract to the Augusta consortium, which comprised CAP (Reading), STL, Ferranti Computer Systems, Scicon, Imperial College department of computing, and the South West Universities Regional Computer Centre (SWURCC).

I was project leader of the Augusta consortium; everything we did was by consensus and peer discussion. Our report, delivered in September 1981, took a few example problems, expressed a design following several different methods, and developed implementations from each in Ada. We also did a literature study of many more design methods and of developers. Among the mostly structured methods (such as JSD), we used and/or considered CCS and VDM.

In about 1978 ITT divested itself of STC, so that STC became an entirely British company. I soon discovered that British management had some unwelcome sides. Meanwhile the South West universities “privatised” their Computer Centre, and SWURCC became the software house, Praxis. Praxis had a quality ethos that was sympathetic to the use of rigorous and formal methods. Three of us from STL/STC moved to Praxis. There we taught the use of VDM and the underlying discrete maths. Others propagated the use of Z, with Mike Spivey's *fu*ZZ tool, notably in work on CICS for IBM. We expanded the audience for our courses on VDM and discrete maths through the National Computer Centre and to some of Praxis' customers.

Formal Methods Europe was formed, initially as Formal Methods Europe, for the first few years with European Commission funding. I must acknowledge the original championship of Karel de Vriendt, of the Commission staff, in this initiative. Eventually the organisation had to become self-sustaining, and after some initial trepidation, this has succeeded.

My seven years with Praxis were followed by secondments to the DTI Information Technology Division, and a long sequence of short contracts with the EC in the successive Framework programmes. The projects with these government bodies only occasionally involved formal methods. Then Lloyd's Register grew its own software research group. Lloyd's Register started life registering ships which were deemed to be seaworthy, simply maintaining a list; they soon moved to assessing the seaworthiness of ships, then diversified further to assessing the safety of a wide range of engineering systems. At some point they woke up to the fact that they were blithely giving safety certificates to engineering systems containing computers, without knowing much about the embedded software. So a new field of work developed at LR: the verification of safety-critical software. The first major contract was static analysis, using the MALPAS suite, of the secondary protection system of the Sizewell B nuclear power station. Another LR project was the safety assessment of a new digitised version of an electromechanical subsystem in the Hercules transport plane.

My final project as a solo consultant was to help Dines Bjørner set up *FORTIA*, the Formal Techniques Industrial Association, in 2003. With my background in company operations, I was able to draft the By-laws and Charter of the organisation, at the same time understanding what the whole enterprise was about, technically and motivationally. Some thirty organisations from fifteen countries were recruited and became its first members.

Telecoms involves much concurrent, real-time processing. At STL some of us experimented with Milner's CCS, but we considered other formalisms too: Petri Nets, CSP, Temporal Logic. At the time Milner's own view of CCS was as a theoretical model of concurrency. How does one compare these formalisms? For what type of problem is each most suited? How do they relate to each other? STL held a workshop in 1983[15], setting nine problems to which solutions were proposed by the expert participants.

15 B. T. Denvir, W. T. Harwood, M. I. Jackson, M. J. Wray, eds.: *The Analysis of Concurrent Systems*, LNCS 207, Springer-Verlag, 1985.

So, to summarise, formal methods started life with formal grammars, which facilitated grammar-driven parser generators, which in turn made language compilers more immediately related to the languages they accepted. Then means of defining semantics of languages came about, λ-calculus, operational and denotational semantics; you can't prove a program correct unless you have a way of defining the meaning of “sentences” of the language, i.e. programs. In software development, a functional specification expressed as the semantics of the desired end-product, i.e. of the program, enables a proof to be constructed of its correctness. Development methods, such as VDM, which evolved from VDL, were thus based on semantics, precisely because of this connection. Then more aspects of the life-cycle began to be expressed in formal terms, notably the requirements.

What of the last ten years? A programming language is a medium through which a user communicates with the computer and instructs it to perform a desired task. More and more, this medium is ceasing to be a linear script of symbols. Even with as banausic an object as a spreadsheet, the “program”, which in this case is an array of expressions, is constructed by means of an interactive and non-linear conversation with the package. The composition of a web page is achieved through an even more non-linear communication, perhaps involving the movement of a mouse. In these activities, one is left with no record of the construction process, which mitigates against any quality procedures, let alone a formal description.

So, just as in the early days when researchers were focusing on syntax instead of semantics, are we once again looking under the wrong lamppost, or barking up the wrong tree?

Acknowledgements

Working with many colleagues and hearing lectures and reading the works of others has been both instructive and a pleasure. I worked closely with Roger Shaw and Mel Jackson in bringing formal methods to STL and Praxis, and with Bernie Cohen at STL. Anthony Hall drove the use of Z in Praxis. Karel de Vriendt, on the European Commission staff, was the local champion who enabled the birth of VDM Europe, which evolved into FME. Roger Shaw influenced the introduction of formal verification for safety-critical software at Lloyd's

Register. On that LR team, Maurice Naftalin used MALPAS for the static analysis of the Sizewell B secondary protection system. My STL colleagues and I learned much from consultations with Cliff Jones, Dines Bjørner, Robin Milner, Bernard Sufrin, and Carrol Morgan. At STL, Paul Taylor and Will Harwood enlarged my understanding of proof techniques. I have already mentioned the profound effect of the 1979 Winter School in Copenhagen and the lecturers at that event. Finally, I have learned much and been particularly inspired by the written works of Edsger Dijkstra, Tony Hoare and Robin Milner.

Tim Denvir

BCS-FACS/LMS Evening Seminar

Joint event with Formal Methods Europe

Thursday 4th May 2017, 6:00pm

Venue: BCS London office, London.

Prof. Dr. Reiner Hähnle

(TU Darmstadt, Germany)

The KeY Formal Verification Tool

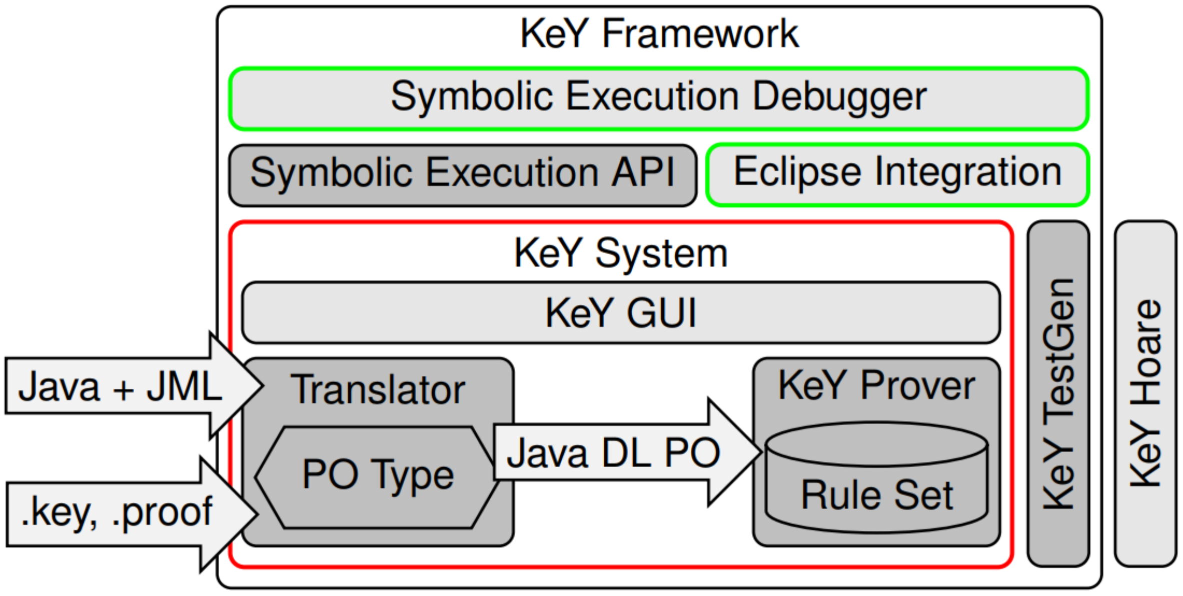
Reported by Jonathan Bowen

Abstract: KeY is a deductive verification tool for sequential Java programs. It is based on a rich program logic for Java source code. KeY can perform functional verification of Java programs annotated with specifications in the Java Modeling Language (JML). Specification elements include class invariants and method contracts. The rules of KeY's program logic realize a symbolic execution engine for Java. Verification proceeds method-wise, unbounded loops are approximated by invariants, method calls by contracts. KeY incorporates state-of-art proof search and an auto-active mode that in many cases results in fully automatic proofs. Otherwise, the user can perform interactive steps or ask the system to search for a counter example. KeY has been successfully used to verify complex legacy code, such as the sort method of the Java Development Kit (JDK), where a subtle bug was found and subsequently fixed. I will explain some of the theoretical underpinnings and design principles of KeY. The talk included a live demonstration of some of KeY's capabilities.

(Photograph by Jonathan Bowen)

Prof. Dr. Reiner Hähnle of the Software Engineering Group at the Technische Universität Darmstadt (TU Darmstadt), a research university in the city of Darmstadt, Germany, gave a talk to members of the BCS and Formal Methods Europe (FME) at the BCS London office in Southampton Street on the evening of 4th May 2017. The talk was preceded by the AGM of FME and was cosponsored by BCS-FACS and FME.

The talk presented the KeY formal verification tool. The KeY framework consists of a number of components, including the KeY system itself, illustrated as follows:



The KeY system is covered by a recent edited book (Ahrendt et al., 2016). The KeY approach provides mechanized support for deductive verification of object-oriented programs. Specifically, a Java program and Java Modeling Language (JML) specification can be provided to a proof obligation (PO) generator to output theorems in dynamic logic (DL), an extension of modal logic for reasoning about computer programs. This can be input to the KeY prover. The KeY tool supports both interactive and fully automated correctness proofs.

A design-by-contract methodology is followed. Contracts are used to specify methods, where preconditions must be established by the caller, postconditions are guaranteed by the called routine if the precondition holds at invocation time, and the behaviour of method calls can be approximated by their contract. Invariants are attached to classes to specify global system properties and data consistency properties. The behaviour of unbounded loops is approximated by their invariant.

The representation of proof obligations in logic can be viewed in different way, for example the sequent calculus view, the program logic view, and the symbolic execution view. Reasoning is undertaken by syntactic transformation using schematic proof rules. Goal-directed proof search is undertaken using the KeY language to describe rule schemata together with application heuristics in the form of "taclets". A demonstration was given during the talk.

The KeY calculus includes different classes of rule schemata, such as logical rules (first-order ground rules, an induction rule, and weakening), simplification rules, theory rules, symbolic execution rules (for each Java statement), and modularity and abstraction using specification annotations.

KeY's automated proof search strategy simplifies aggressively after each symbolic execution (SE) step. The advantages of a logic embedding of symbolic execution include: early pruning of unreachable statements, unlike a verification condition generator (VCG); keeping path conditions and the symbolic state as simple as possible; providing sufficient information for white-box test generation and symbolic state debugging; a formal notion of soundness (unlike native symbolic execution); the provision of relational properties as well as correctness (unlike Hoare triples).

KeY provides two modes of proof interaction: 1) interactive rule application (for experts who understand the proof situation well); 2) patching of the specification and/or code after a failed proof (an "auto-active" mode). The latter is closer to the workflow using a model checking approach. In the auto-active workflow, the user starts with a fully automatic proof search. The proof search stops with unprovable subgoals, to a given limit with respect to the number of rule applications reached. The user can then view the SE tree/memory, generate a model, or patch and/or complete the specification and code. KeY's Eclipse extensions support the auto-active mode with background proofs.

Deductive functional verification of algorithmically complex programs requires expert interactions. It is time-intensive and thus expensive. However, the correctness of library functions is crucial since these are used in many programs. For example, the Java library sort and binarySearch functions used to include software bugs.

An extensive example of “Timsort”, a hybrid sorting algorithm using insertion sorting and merge sorting, was presented. This algorithm is optimized for partially sorted arrays, as often encountered in real-world data. It was implemented using Python by Tim Peters in 2002. Since Java 1.7, it has been the default algorithm for non-primitive arrays. Timsort is in the Java standard library used by Oracle, the Python standard library used by Google, the Android standard library also used by Google, and many more languages/frameworks, including Apache. The algorithm was presented in some detail during the talk, including the main loop. The invariant must be re-established in the merge phase of the algorithm.

TimSort allocates a fixed stack size depending on input array size. However, sometimes the allocated length is less that the required stack size for large array sizes. This can generate exceptions in practice. Using the KeY tool revealed issues in re-establishing the invariant and also the allocated stack sizes for large arrays. These were fixed and the library program was re-proved with a new invariant. The class invariant was formally specified and contracts were specified for all methods. Loop invariants were also specified, especially for the merge stage. For each method, it was verified that the contract was satisfied and the class invariant was preserved. The fixed version of the software was formally verified, in contrast to previous fixes. Aspects not yet proven include the sortedness and permutation of the result.

The impact of the Timsort case study on the KeY tool included adding support for bitwise operation and integration of state merging techniques to avoid state explosion of the SE size. It is conjectured that design for verification would decrease the proof effort. Modular method design provides simple and clear contracts, avoidance of complex intra-method control flow, and no reliance on integer overflow. The Timsort bug affected many programming languages and frameworks, including: Java (Oracle JDK, Android), Python, Apache (Lucene, Hadoop, Spark++), Go, D, and Haskell. The input array size needed to trigger the error was 2^{16} for Android, 2^{26} for Java, and 2^{49} for Python.

As a result of this verification effort, Oracle fixed the bug by increasing stack size, based on informal worst case analysis, but with no formal proof. The Python community quickly adopted the formally proven fix. Android implemented a different fix that was also verified with KeY. In conclusion, formal verification can be effective for real-life, complex, mainstream code. The publicity resulted in several hundred thousand page views and being top news on sites such as Reddit, Hacker News, etc. A student commented on Reddit:

"Well, would you look at that. KeY is actually used for something useful. I thought they just tortured us with it for fun at university."

Tim Peters, inventor of Timsort, commented via Python-Bugtracker:

"Some researchers found an error in the logic of merge collapse, explained here, and with corrected code shown in … It should be fixed anyway, and their suggested fix looks good to me."

Joshua Bloch tweeted:

"Congratulations to Stijn de Gouw et al. for finding and fixing a bug in TimSort using formal methods!"

In conclusion, the Timsort study is a well-publicised example of the effective use of formal methods using the KeY tool in detecting and correcting errors in a mainstream and widely used piece of software,

References

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., and Ulbrich, M. (eds.) (2016), Deductive Software Verification – The KeY Book: From Theory to Practice. Springer.

The KeY system can be downloaded and installed from: http://www.key-project.org

For further BCS-FACS information on the talk, including a copy of the slides with further links and papers, see: http://www.bcs.org/content/ConWebDoc/57115

Conquering the Barriers to Formal Specification: Some recent developments in iUML- B and Event-B

Thursday 15 June 2017.

BCS, 1st Floor, The Davidson Building, 5 Southampton Street, London, WC2E 7HA

Colin Snook, University of Southampton, UK

Abstract/Synopsis

iUML-B is a diagrammatic front end for Event-B that was initially conceived in my PhD “Exploring the Barriers to Formal Specification”. My exploration concluded that mathematics is no more difficult to understand than programming languages, but finding the best way to model things is. The aim of UML-B was to encourage industry into formal modelling by making it more visual, approachable and easier to explore different abstractions.

Over the intervening 16 years we have re-developed UML-B several times to reach its current integrated (hence the i) form. We are now using iUML-B and Event-B with industry on a regular basis both for industry-led research projects such as Enable-S3 and for direct contracts with industry. In some of these contracts we are developing the tool support for requested features and in others to develop the technology readiness level. It is probably too soon to say that we have conquered those barriers but I certainly feel that we have achieved a high level of interest. In this talk I will give a brief history of UML-B, summarise our recent activities and plans, and then focus in more depth on one application arising from the Enable-S3 project; analysing security flaws.

Presentation

- Conquering the Barriers to Formal Specification – Colin Snook bcs The Chartered Institute for IT — Formal Aspects of Computing Science Specialist Group

LONDON MATHEMATICAL SOCIETY EST. 1865

BCS-FACS Evening Seminar
Joint event with the London Mathematical Society

Thursday 2 November 2017 6:00 pm

Professor Erika Ábrahám
(RWTH Aachen University)

Symbolic Computation Techniques in SMT Solving

The satisfiability problem is the problem of deciding whether a logical formula is satisfiable. For first-order arithmetic theories, in the early 20th century some novel solutions in form of decision procedures were developed in the area of Mathematical Logic. With the advent of powerful computer architectures, a new research line of Symbolic Computation started to develop practically feasible implementations of such decision procedures.

Independently, for checking the satisfiability of propositional logic formulas, around 1960 a new technology called SAT solving started its career. Despite the fact that the problem is NP complete, SAT solvers showed to be very efficient when employed by formal methods for verification. Motivated by this success, the power of SAT solving for Boolean problems had been extended to cover also different theories. Nowadays, fast SAT-modulo-theories (SMT) solvers are available also for arithmetic problems. These sophisticated tools are continuously gaining importance, as they are at the heart of many techniques for the analysis of programs and probabilistic, timed, hybrid and cyber-physical systems, for test-case generation, for solving large combinatorial problems and complex scheduling tasks, for product design optimisation, planning and controller synthesis, just to mention a few well-known areas.

Due to their different roots, Symbolic Computation and SMT solving tackle the satisfiability problem differently, offering potential for combining their strengths. This talk will provide a general introduction to SMT solving and decision procedures for non-linear arithmetic, and show on the example of the Cylindrical Algebraic Decomposition method how algebraic decision procedures, rooted in Symbolic Computation, can be adopted in the SMT solving context to synthesise beautiful novel techniques for solving arithmetic problems.

Venue: London Mathematical Society, De Morgan House, 57-58 Russell Square, London WC1B 4HS.
Refreshments will be available from 5.30pm.
The seminar is free of charge. If you would like to attend, please email lmscomputerscience@lms.ac.uk.

BCS-FACS

Peter Landin Semantics Seminar 2017

BCS, 5 Southampton Street London, WC2E 7HA

Tuesday 12 December, 6 p.m.

(Tea/coffee from 5:15pm, Drinks reception from 7:15pm - 8:30pm)

Compiling without continuations

Prof. Simon Peyton Jones, FRS

(Microsoft Research)

Abstract:

GHC compiles Haskell via Core, a tiny intermediate language based closely on the lambda calculus. Almost all GHC's optimisations happen in Core, but until recently there was an important kind of optimisation that Core really did not handle well. In this talk I'll show you what the problem was, and how Core's new "join points" solve it simply and beautifully, by effectively allowing Core to express control flow as well as data flow; there are strong links to so-called "continuation passing style" (CPS) here.

Understanding join points can help you as a programmer too, because you can write code confident that it will optimise well. I'll show you a rather compelling example of this: "skip-less streams" now fuse well, for the first time, which allows us to drop the previous (ingenious but awkward) workarounds.

Booking: https://events.bcs.org/book/2701/

FACS Committee

bcs The Chartered Institute for IT — Formal Aspects of Computing Science Specialist Group

Jonathan Bowen
FACS Chair; BCS Liaison

Jawed Siddiqi
FACS Treasurer

Paul Boca
FACS Secretary

Roger Carsley
Minutes Secretary

John Cooke
LMS Liaison

Ana Cavalcanti
FME Liaison

Margaret West
BCS Women Liaison

Rob Hierons
Chair, Testing Subgroup

John Derrick
Chair, Refinement Subgroup

Eerke Boiten
Chair, Cyber Security Subgroup

Sofia Meacham
Meetings Coordinator

Mike Hinchey
International Coordinator

Tim Denvir
Co-Editor, FACS FACTS

Brian Monahan
Co-Editor, FACS FACTS

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

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

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

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.