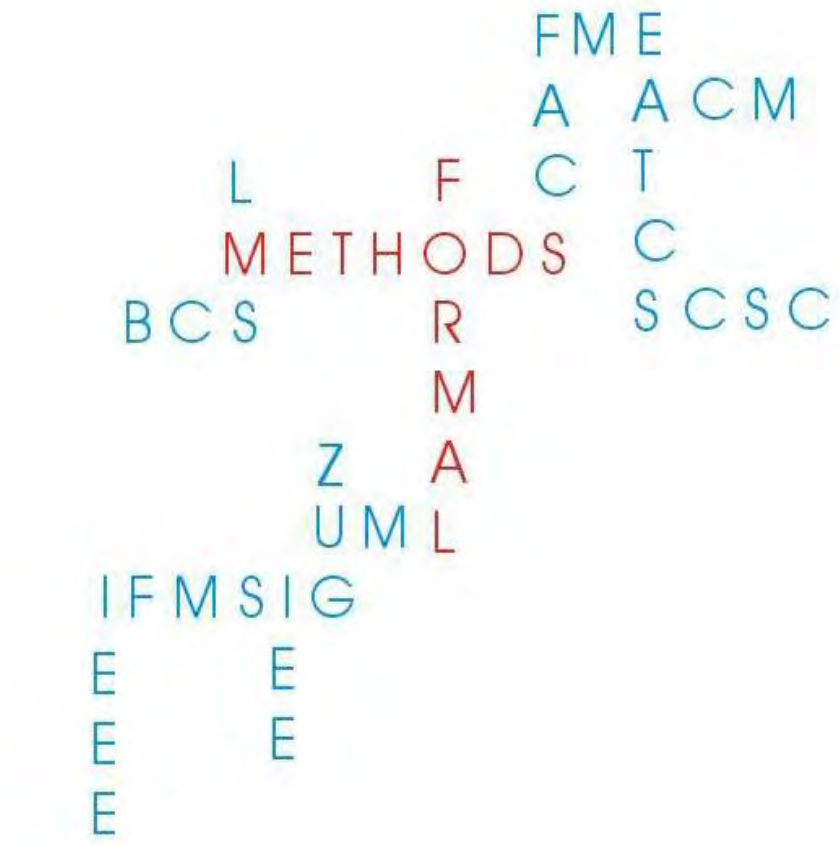
Issue 2018-1
August 2018

FACS ACTS



bcs The Chartered Institute for IT
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, John Cooke, Tim Denvir, Brian Monahan, Margaret West.

Contributors to this issue

Jonathan Bowen, John Cooke, Tim Denvir, Sofia Meacham.
Brian Monahan, Bill Stoddart, Botond Virginas, 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

Dear readers, welcome to our first issue of *FACS FACTS* for 2018.

This year, 2018, marks the 40th anniversary of FACS. At least one editor recalls an article by Dan Simpson, member of the editorial team at the time, *FACS at 10* in 1988. That 10th anniversary seemed momentous then, and now it feels incredible that another 30 years have passed. The 20th anniversary was marked with a major event at the Royal Society involving four Fellows of the Royal Society as speakers: Mike Gordon, Tony Hoare, Robin Milner, and Gordon Plotkin. Sad to say that two of these eminent computer scientists are no longer with us. Robin Milner passed away in 2010 and Mike Gordon, only recently and at far too young an age of 69 on 22 August 2017, almost exactly a year ago.[1] Both worked at the University of Edinburgh (where Mike Gordon undertook his PhD degree with Rod Burstall) and the University of Cambridge.

We are not marking this 40th anniversary occasion quite so grandly as the 20th, but on 12 October 2018 we plan to celebrate the 20th anniversary of the publication of the book *Unifying Theories of Programming* by Prof. Tony Hoare and Prof. He Jifeng. Both the original authors will be present, together with Prof. Jim Woodcock who has undertaken much UTP research in the intervening period. Tony Hoare will provide some introductory remarks, He Jifeng, travelling from East China Normal University in Shanghai, will give the main talk on *Unifying Theories of Refinement*, and Jim Woodcock of the University of York will sum up at the end. We also plan to celebrate the 40th anniversary of FACS during this event as well, with teatime networking before and a buffet reception after the main talk. We have booked extra space for this event and hope that as many FACS members as are able will attend. With celebrations for the 10th, 20th, and 40th anniversaries, we also look forward to the 80th anniversary!

Earlier in the year, we received a paper from Bill Stoddart on *The Halting Paradox*, which is the first item in the present newsletter. This paper stimulated much discussion amongst the committee. We are not a peer-reviewed journal, but we nonetheless wish to retain some degree of academic status, and we debated amongst ourselves how to treat submissions of technical papers. We decided to present Bill Stoddart's paper as a starter for discussion, and it appears below, as submitted. For example, how does it relate to Scott Domain theory, fixpoints, three-valued logic? And to Turing's approach, which relies on recursive enumerability? We invite and strongly welcome well-reasoned responses from readers – perhaps to appear in the next *FACS FACTS* newsletter!

The second item is *Autonomics and their Verification* from BT's industrial perspective by Sofia Meacham, Bournemouth University, and Botond Virginas,

[1] For an excellence account of Mike Gordon's life and work, see: Lawrence C. Paulson (11 June 2018) *Michael John Caldwell Gordon (FRS 1994), 28 February 1948 – 22 August 2017*, arXiv:1806.04002.

BT Adastral Park. The authors offer this as a position paper and would also like to invite discussion.

After this, we have reports on several FACS events since the last issue, preceded by photographs of the speakers, taken by Jonathan Bowen: the joint FACS-LMS Seminar *Symbolic Computation Techniques in SMT Solving*, Prof. Erika Abraham, University of Aachen, report by Tim Denvir; *Compiling without Continuations*, Prof. Simon Peyton Jones FRS, Microsoft Research, report by Margaret West and Brian Monahan; *The Fumble Programmer*, Roderick Chapman, University of York, report by Sofia Meacham; *Model-Based Testing of Cyber-Physical Systems*, Mohammad Mousavi, University of Leicester, report by John Cooke; and a photographic account from Jonathan Bowen of the FME Fellowship Award at the FM 2018 International Symposium on Formal Methods in Oxford this year, which BCS-FACS sponsored. Finally, we include a book review of *Modeling Companion for Software Practitioners*, by Egon Börger and Alexander Raschke, published by Springer, 2018, reviewed by Jonathan Bowen.

We have several forthcoming events during the rest of the year. On this, we have already mentioned the UTP 20th anniversary seminar by He Jifeng on 12 October 2017. We also have a speaker from the National Physical Laboratory (NPL), Stephane Chretien, on 17 October 2018, co-organised by Sofia Meacham and Keith Lines. We welcome two new FACS committee members, Keith Lines of NLP and Mohammed Mousavi of University of Leicester. A highlight of the year for FACS is the joint FACS-LMS seminar, now organized by Rob Hierons, this year to be given by Prof. Bill Roscoe of the University of Oxford on 1 November 2018. Our most significant annual event is the Peter Landin Semantics seminar, organized as ever by Paul Boca, this year delivered by Prof. Don Sannella of the University of Edinburgh on 10 December 2018. This will be preceded by the FACS AGM and members of FACS are especially encouraged to attend both.

Most FACS seminars take place in the offices of the BCS in the Davidson Building, Southampton Street, close to Covent Garden underground station. The FACS-LMS joint seminar is held at the headquarters of the London Mathematical Society (LMS) at De Morgan House, 57-58 Russell Square, London. The nearest underground station is Russell Square.

A special thank you to the co-editor of the *FACS FACTS* newsletter, Brian Monahan, for his excellent work on ensuring that it is so well presented overall. We hope you enjoy this issue and welcome contributions for future issues.

Tim Denvir, *FACS FACTS* co-editor
Jonathan Bowen, FACS Chair

The Halting Paradox

Bill Stoddart

December 20, 2017

Abstract

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has been "proved" in many text books and taught on many computer science courses, and is supposed to illustrate the limits of computation. However, Eric Hehner has a dissenting view, in which the *specification* of the halting problem is called into question.

Keywords: halting problem, proof, paradox

1 Introduction

In his invited paper [1] at The First International Conference on Unifying Theories of Programming, Eric Hehner dedicates a section to the proof of the halting problem, claiming that it entails an unstated assumption. He agrees that the halt test program cannot exist, but concludes that this is due to an inconsistency in its specification. Hehner has republished his arguments using less formal notation in [2].

The halting problem is considered to be an essential part of the theoretical background to computing. That halting is not in general computable has been "proved" in many text books and taught on many computer science courses to illustrate the limits of computation. Hehner's claim is therefore extraordinary. Nevertheless he is an eminent and highly creative computer scientist whose opinions reward careful attention. In judging Hehner's thesis we take the view that to illustrate the limits of computation we need a consistent specification which cannot be implemented.

For our discussion we use a guarded command language for sequential state based programs. Our language includes named procedures and named enquiries and tests. These constructs may have input parameters. Enquiries and tests allow assignment only to their own local variables; they are thus side effect free. They are used in expressions, where they represent the value returned by the enquiry. Procedures must be invoked as program statements.

With the text of each program P we associate a unique number $\lceil P \rceil$, known as the program's encoding, which will stand for the program when we want to use that program as data, e.g. when passing one program to another as an argument.

The halting problem is typically stated as follows. Given a Turing machine equivalent (TME) language there is no halt test program $H(\lceil P \rceil, X)$ which will tell us, for arbitrary program P and data X, whether P will halt when applied to X.

Hehner simplifies this, saying there is no need to consider a program applied to data, as data passed to a program could always be incorporated within the program. So his version is that there is no halt test $H(\lceil P \rceil)$ which tells us, for an arbitrary program P, whether execution of P will halt.

Where context allows us to distinguish P from $\lceil P \rceil$, we may allow ourselves to write P instead of $\lceil P \rceil$. For example we will write $H(P)$ and $H(P, X)$ etc rather than $H(\lceil P \rceil)$, $H(\lceil P \rceil, X)$.

The proof that H cannot be implemented goes as follows. Under the assumption that we have implemented H, and that we have a program *Loop* which is a non-terminating loop, we ask whether the following program will halt:

$S \mathrel{\widehat{=}}$ **if** $H(S)$ **then** *Loop* **end**

Now within S, H must halt and provide a true or false judgement for the halting of S. If it judges that S will halt, then S will enter a non-terminating Loop. If it judges that S will not halt, then S will halt.

Since H cannot pass a correct judgement for S, we must withdraw our assumption that there is an implementation of H. Thus halting behaviour cannot, in general, be computed. □

Hehner asks us to look in more detail at the specification of H. If we can specify H there will be some interest in proving it is not implementable. If we cannot specify it we have a problem: we won't be able to say formally what it is that cannot be implemented.

The paper is structured as follows. In section 2 we verify Hehner's simplification of the halting problem. In section 3 we make some general remarks on unbounded memory calculations and connections between halt tests and mathematical proofs. We also show that failure to halt is observable for computations with fixed memory resources. In section 4 we consider a halt test that is required to work only for a small set of stateless programs, and we find we can still use the same proof that we cannot have a halt test. We examine the specification of the halt test for this limited scenario in detail, and we describe an implementation of an amended halt test that is allowed to report non-halting by an error message if the test itself cannot terminate. In section 5 we perform a semantic analysis of S, taking its definition as a recursive equation, and conclude that its defining equation has no solution. S does not exist as a conceptual object, and neither does our putative halt test H.

The halting problem is generally attributed to Turing's paper on Computable Numbers [3], but the connection is slightly less direct than this implies. In an appendix we briefly describe Turing's paper and how the halting problem emerged from it. We give an example, from Turing's paper, of an uncomputable function which has a consistent specification.

2 Hehner's simplification of the halting problem

Normally the halting problem is discussed in terms of a halt test taking data D and program P and reporting whether P halts when applied to D.

Hehner's simplified halt test takes a program P and reports whether it halts.

We refer to the first of these halt tests as H_2, since it takes two arguments, and the second as H.

To verify Hehner's simplification of the halting problem we show that any test that can be performed by H_2 can also be performed by H, and any test that can be performed by H can also be performed by H_2.

Proof Given procedures P_0 and $P_1(x)$, and tests $H(p)$ and $H_2(p, d)$ where $H(P_0)$ reports whether P_0 halts, and $H_2(P_1, d)$ reports whether $P_1(d)$ halts:

then if we define an operation $T \mathrel{\widehat{=}} P_1(d)$ the test $H_2(P_1, d)$ can be performed using H as $H(T)$.

and if we define an operation $U(x) \mathrel{\widehat{=}} P_0$, where the name x is chosen to be non-free in P_0, the test $H(P_0)$ can be performed by H_2 as $H_2(U, d)$. □

3 Some notes on halting analysis

Fermat's last theorem states that for any integer $n > 2$ there are no integers a, b, c such that:

$$a^n + b^n = c^n$$

Since 1995, when Andrew Wiles produced a proof 150 pages long, the theorem is recognised as true.

Given a program *Fermat* which searches exhaustively for a counter example and halts when it finds one, and a halt test H we could have proved the theorem by executing the test $H(Fermat)$ This would tell us the program *Fermat* does not halt, implying that the search for a counter example will continue forever, in other words that no counter example exists and the theorem is therefore true.

The Goldbach conjecture, which states that every even integer can be expressed as the sum of two primes (we include 1 in the prime numbers). This is an unproved conjecture, but so far no counter example has been found, although it has been checked for all numbers up to and somewhat beyond 10^{18}. Given a program *Goldbach* which searches exhaustively for a solution to the Goldbach conjecture and terminates if it finds one, and halt test H for *Goldbach*, we could decide the truth of the conjecture by executing the test $H(Goldbach)$.

The programs *Fermat* and *Goldbach* would have unbounded memory requirements, as they must be able to deal with increasingly large integers.

When considering the halting problem we normally assume idealised computers with unbounded memory. But suppose we have a program that is to be run on a computer with n bits of memory. Its state transitions can take it to at most 2^n different states. We can solve the halting problem for this program by providing an additional n bits of memory and using this as a counter. When we have counted 2^n state transitions and the program has not halted, we know it will never halt because it must have, at some point, revisited a previous state.

Of course the question being answered by the proof, on the one hand, and the

monitoring of execution, on the other, are not the same. Monitoring execution does not require the "twisted self reference" (Hehner's term) that occurs in our proposed program S. There is a separation between the monitor, as observer, and the executing program, as the thing observed.

4 A halt test for a limited set of programs

The conventional view of the halting problem proof is that it shows a universal halt test is impossible in a TME language. In this section we seek to clarify the inconsistency of the halt test specification by limiting our discussions to a small set of state free programs.

Consider first the set $\mathcal{L}_0 = \{ \lceil Skip \rceil , \lceil Loop \rceil \}$

$\mathcal{L}_0$ is a set for which we can specify a halt test H_0. The specification is consistent because it has a model:

$$\{ \lceil Skip \rceil \mapsto true , \lceil Loop \rceil \mapsto false \}$$

Now we become ambitious and wish to consider the set:

$\mathcal{L}_1 = \{\lceil Skip \rceil, \lceil Loop \rceil, \lceil S \rceil\}$, with a halt test H.

Our definition of S is still:

$$S \mathrel{\widehat{=}} \textbf{if} \quad H(S) \quad \textbf{then} \quad Loop \quad \textbf{end}$$

and our specification for H is:

For $p \in \mathcal{L}_1$, $H(p)$ is true if execution of p halts, and false otherwise.

But what is the model for H ?

$$\{ \lceil Skip \rceil \mapsto true , \lceil Loop \rceil \mapsto false, \lceil S \rceil \mapsto ? \}$$

Our model must map S to either true or false, but whichever is chosen will be wrong. We have no model for H, so it cannot have a consistent specification.

We have reduced the halting problem to this limited scenario so we can attempt to write out the model of halting, but exactly the same argument applies to halting in a TME language.

In this limited scenario we can make the same "proof" that halting is uncomputable that we used for TME languages in the introduction.

4.1 Experiments with code

Setting aside formal analysis for a moment, let us see what our programming intuition can tell us about the strange program

$S \;\widehat{=}\; \textbf{if}\;\; H(S)\;\; \textbf{then}\;\; \mathit{Loop}\;\; \textbf{end}\,.$

And let us see if we can tweak H so that S can actually be implemented.

Although the halt test is unable to tell us this, S looks as if it will NOT terminate, because when $H(S)$ is executed within S, it will be faced with again deciding the result of $H(S)$ with no additional information to help it. S will not terminate, but this is because the halt test invoked within it cannot terminate.

There is no reason, however, why the halt test cannot terminate in other situations, or why failure to halt cannot be reported via an error message when the halt test itself cannot halt.

We now consider a halt test $H_1(p)$ for use with the set

$\mathcal{L}_2 \;=\; \{\, \lceil \mathit{Skip} \rceil,\; \lceil \mathit{Loop} \rceil,\; \lceil S_1 \rceil \,\}$

where: $S_1 \;\widehat{=}\; \textbf{if}\;\; H_1(S_1)\;\; \textbf{then}\;\; \mathit{Loop}\;\; \textbf{end}$

For $p \;\in\; \mathcal{L}_2$, $H_1(p)$ returns a true flag if execution of p halts. If execution of p does not halt return a false flag, unless H_1 has been invoked within S_1, in which case report an error.

Here is the error report when S_1 is invoked.

Error at S_1
Cannot terminate
reported at H_1 *in file ...*

Implementation of H_1 requires it to determine whether it is being invoked from within S_1. In a typical compiled sequential language this information can be deduced from the return address for the call to H, and the symbol table, which contains information that will tell us whether this return address is within the code body of S_1. However, this information is not usually directly accessible in the language, so we will suppose we have written at assembly code level a test $\mathit{In}S_1$ which will report whether the operation that invokes it has been invoked from within S_1.

We also assume an error handler $\mathit{Error}(s)$ which is invoked when an error condition is detected. It prints the string s as an error message, and handles the error. Formally this a form of non-termination.

In defining H_1 we build into it the conclusions we deduced above: S_1 does not terminate, but H_1 cannot report this in the normal way if it has been invoked from within S_1.

$H_1(p) \mathrel{\widehat{=}}$
  if $p = Skip$ **then** $return(TRUE)$
  elseif $p = Loop$ **then** $return(FALSE)$
  elseif $p = S_1$ **then**
    if InS_1 **then** $Error(\text{“}Cannot\ terminate\text{”})$ **else** $return(FALSE)$ **end**
  else $Error(\text{“}Invalid\ program\text{”})$
  end

This illustrates that the problem is not that halting of S_1 cannot be computed, but that the result cannot always be communicated in the specified way. Requiring H (or in this case H_1) to halt in all cases is too strong, as it may be the halt test itself that cannot halt.

5 Proof and paradox

In [1] the halting problem is compared to the Barber's paradox. "The barber, who is a man, shaves all and only the men in the village who do not shave themselves. Who shaves the barber?" If we assume he shaves himself, we see we must be wrong, because the barber shaves only men who do not shave themselves. If we assume he does not shave himself, we again see we must be wrong, because the barber shaves all men who do not shave themselves. The statement of the paradox seems to tell us something about the village, but it does not, since conceptually no such village can exist.

In a similar way, the program S which we have used in the halting problem proof, does not exist as a conceptual object so what we say about it can be paradoxical.

To argue this formally we use the following termination rule:

$$trm(g) \Rightarrow (trm(\textbf{if } g \textbf{ then } T \textbf{ end}) \Leftrightarrow \neg g \vee (g \Rightarrow trm(T))) \quad (1)$$

And we specify the result of applying H to program P as:

$$H(P) \Leftrightarrow trm(P)$$

Bearing in mind that $trm(H)$ is true by the specification of H, we argue:

$trm(S) \Leftrightarrow$ " by definition of S "
$trm($ **if** $H(S)$ **then** $Loop$ **end** $) \Leftrightarrow$ " by rule (1) above "
$\neg\, H(S) \vee (\, H(S) \Rightarrow trm(Loop)\,) \Leftrightarrow$ " property of $Loop$ "
$\neg\, H(S) \vee (\, H(S) \Rightarrow false\,) \Leftrightarrow$ " logic "
$\neg\, H(S) \vee \neg\, H(S) \Leftrightarrow$ " logic "
$\neg\, H(S) \Leftrightarrow$ " specification of H "
$\neg\, trm(S)$

So we have proved that $trm(S) \Leftrightarrow \neg\, trm(S)$. This tells us that S does not exist as a conceptual object, let alone as a program. We have seen in the previous section that by relaxing the specification of H we can implement the same textual definition of S, so the non existence of S proved here can only be due to the specification of H being inconsistent.

The proof of the halting problem assumes a universal halt test exists and then provides S as an example of a program that the test cannot handle. But S is not a program at all. It is not even a conceptual object, and this is due to inconsistencies in the specification of the halting function. H also doesn't exist as a conceptual object, and we have already seen this from a previous argument where we show it has no model.

6 Conclusions

The idea of a universal halting test seems reasonable, but cannot be formalised as a consistent specification. It has no model and does not exist as a conceptual object. Assuming its conceptual existence leads to a paradox.

The halting problem is universally used in university courses on Computer Science to illustrate the limits of computation. Hehner claims the halting problem is misconceived. Presented with a claim that a universal halt test cannot be implemented we might ask – what is the specification of this test that cannot be implemented? The informal answer, that there is no program H which can be used to test the halting behaviour of an arbitrary program, cannot be formalised as a consistent specification.

The program S, used as example of a program whose halting cannot be analysed, observes its own halting behaviour and does the opposite. Hehner calls this a "twisted self reference". It violates the key scientific principle of, where possible, keeping what we are observing free from the effects of the observation.

To better understand Hehner's thesis we have verified his simplification of

the halting problem, and studied a set of three stateless programs and a halt test, to which exactly the same proof can be applied.

Our programming intuition tells us that S will not terminate because when $H(S)$ is invoked within S, H will not terminate. However, we cannot require H to return a value to report this, because that would require it to terminate! We provide a programming example based on a a halt test for a small set of programs, where we resolve this by allowing the option for the halt test to report via an error message when it finds itself in this situation. However, we can require that the halt test should always halt in other situations. The problem, in our little scenario for which the halting problem proof can still be applied, is not the uncomputability of halting!

We have also performed a semantic analysis which confirms that the halt test and S *do not exist as conceptual objects*.

Having a halt test for specific unbounded memory computations, such as those that search for counter examples to Fermat's last theorem and the Goldbach conjecture, involves no inconsistency, and would give us enormous mathematical powers, but this has nothing to do with the halting paradox.

We have found nothing to make us disagree with Hehner's analysis. Defenders of the status quo might say – so the halt test can't even be conceived, so it doesn't exist. What's the difference? Hehner says that uncomputability should refer to what can be defined (specified) but not implemented. Turing's uncomputable sequence β provides such an example, and is discussed in the appendix.

Acknowledgements.

Thanks to Ric Hehner for extensive electronic conversations and to Steve Dunne for extended discussions.

References

[1] E C R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S E Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[2] E C R Hehner. Problems with the halting problem. *Advances in Computer Science and Engineering*, 10(1):31–60, 2013. See www.cs.utoronto.ca/ hehner/halting.html.

[3] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

Appendix: Turing's 1936 paper and the halting problem

On computable numbers, with a contribution to the Entscheidungsproblem, Turing's paper from 1936 [3] is cited as the source of the halting problem, but it does not use the actual term "halting". The paper captures Hilbert's notion of an "effective procedure" by defining "computing machines", consisting of finite state machines with an infinite tape, which are similar to what we now call Turing machines but with significant differences. He uses such machines to define all numbers with a finite representation as "computable numbers", with the fractional part of such a number being represented by a machine that computes an infinite binary sequence. The description of these machines is finite, so numbers such as π, which are computable to any desired accuracy, can have a finite representation in terms of the machines that compute them.

Turing's idea of a computer calculating π would perhaps have been of a human being at a desk, performing the calculation, and now and then writing down another significant figure. His "computing machines" are supposed to continue generating the bits of their computable sequence indefinitely, but faulty machines may fail to do so, and these are not associated with computable sequences.

The computing machines that generate the computable sequences can be arranged in order using an encoding method which yields a different number for each computing machine.

The computable sequences define binary fractions that can be computed. Turing's contribution to the Entscheidungsproblem is in *defining* a binary sequence β that *cannot be computed*. Let $M(n)$ be the *nth* computable sequence, and define the sequence:

$\beta(n) = \quad \textbf{if } M(n)(n) = 1 \textbf{ then } 0 \textbf{ else } 1 \textbf{ end}\,.$

By a diagonalisation argument β is not one of the computable sequences: it is definable but not computable. The link with halting comes from asking why it cannot be computed, the reason being that although we can talk about the sequence of computing machines that generate infinite binary sequences of 0's and 1's we cannot distinguish these from machines which have the correct syntactic properties but which do not generate infinite sequences. So we cannot compute which of the computable sequences is the nth computable

sequence because we cannot distinguish good and bad computing machines.

The first reference to the "halting problem" I have been able to find comes in M Davis *Computability and Unsolvability*, from 1958. By then Turing machines had taken their current form and were required to halt before the output was read from their tape. He credits Turing's 1936 paper as the source of the problem's formulation.

A proof using a computing mechanism which enquires about its own halting behaviour and then does the opposite appears in M Minsky, *Computation. Finite and infinite machines*, from 1967. This proof, with reference to Minsky, is also given in R Feynman *Lectures on computation.*

Text books that use a programming notation (rather than Turing machines) to discuss the halting problem include *Computer Science, a modern introduction* By L Goldschlager and M Lister, from the Prentice Hall red book series, *Introduction to Computer Science* by V Zwass, and *Discrete Mathematics with Proof* by E Godssett.

Autonomics and their verification from BT's industrial perspective

Sofia Meacham, Botond Virginas
[1] Faculty of Science and Technology, Bournemouth University,
Fern Barrow, Poole, Dorset, BH12 5BB, UK
smeacham@bournemouth.ac.uk,
[2] BT Adastral Park, UK
botond.virginas@bt.com

Abstract. In this position paper, autonomic applications and their requirements as regards to verification and validation will be presented and detailed from BT's industrial perspective. First of all, an overview and the history of AI algorithms and autonomics will be explained and the path from algorithms to systems will be introduced. Then, "business" autonomics will be specified and the realization that we need to move from static algorithms to dynamic ones is particularly emphasized. Software engineering approaches such as Systems Modeling and Verification & Validation can benefit this process and ensure the "correct" transition of autonomic algorithms to their applicability to business contexts, and therefore contribute to their adoption and commercialization.

Keywords: autonomics, verification & validation, systems engineering

1 Introduction

This position paper will present the problems and challenges that arise from BT's industrial perspective as regards to their design, development and adoption of autonomic systems.

BT's autonomic team has been working on developing new AI algorithms and creating patents for their methods for a long time now. The complexity and the know-how for these systems has always been distributed through the knowledge and expertise of their engineers and most of the times documentation of the whole process and the systems involved is minimal to non-existent. The increased complexity of the emerging Big data systems has created more problems due to the lack of appropriate design and

documentation. AI related properties such as stability, robustness, explainability have emerged due to the dynamic nature of autonomic systems.

For all these reasons, it has been identified that software engineering methods would benefit the above process. Especially, new software engineering methods need to be developed to tackle the autonomic systems as regards to their design, development and adoption.

The remainder of the paper will cover in section 2 a chronological approach to the autonomic systems. In section 3, BT's industrial perspective is presented. Specifically, in section 3.1 the transition from static algorithms to dynamic systems with controls is described and in section 3.2 the autonomics system design is provided. Section 4 presents the V&V challenges for these systems: Properties Verification, and section 5 offers conclusions and suggestions for future work and research directions.

2 Autonomic systems: a chronological approach

Since the 1960's the topic of system adaptivity has been extensively studied and the basic principles of self-adaptivity have been put into practice in several application areas. It is no science fiction any more that the systems of the future will be self-adaptive, self-learning, self-healing, self-organizing and any self-related properties that we can define to accommodate for the increasing demand for intelligent systems and the consequences of their adoption.

Macias-Escriva et al provide a comprehensive survey of self-adaptivity past and present together with associated tools and methods. They conclude that although more and more advanced AI techniques are being employed, the emphasis is on more and more sophisticated machine learning techniques in an open loop structure setting rather than a closed loop system. The incorporation of closed-loop mechanisms into such software systems is imperative, so that they can adapt themselves to changing conditions [2].

A major breakthrough in this field came with IBM's autonomic computing initiative [3]. According to IBM's autonomic blueprint self-managing capabilities in a system accomplish their functions by taking an appropriate action based on one or more situations that they sense in the environment. The function of any autonomic capability is a control loop that collects details from the system and acts accordingly. An autonomic manager implements a control loop that accesses and controls a single or multiple managed resource that exists in the run-time environment of an IT system. IBM's white paper organizes these control loops into four categories: self-configuring, self-healing, self-optimizing and self-protecting. One of the key features of autonomic control is that

adaptable policy—rather than hard-coded procedure—determines the types of decisions and actions that autonomic capabilities perform.

Many papers have been published since the IBM manifesto (Mcann and Huebscher [4] present a thorough review of the field) dealing with various aspects of the development, analysis and validation methods for autonomic systems. One of the conclusions from these studies is that there isn't yet enough focus on the feedback loops and their associated properties in order to control self-adaptation in an autonomic system. Understanding and reasoning about the feedback loop is key for building self-adaptive systems from an ad-hoc trial-and-error endeavor towards a more systematic, disciplined approach.

3 BT's industrial perspective

3.1 From static algorithms to dynamic systems with controls

From BT's autonomics team long lasting experience, several observations regarding the requirement for a transition from static algorithms to dynamic systems were made. Static software systems have the inherent problem that after a period of time they may no longer be fit for purpose. Underlying data may have changed, or the environment may have changed, or the priorities may have changed. Complexity might grow around the software and it can be very costly to review and be costly to change. Therefore, static algorithms can be very difficult to sustain and maintain over a period of time.

For this purpose and internally at BT, a simulator has been built around this problem. The simulator is converting an existing BT process comprising of a machine learning algorithm used to make a business decision leading to binary actions and capturing the output in the form of failures and costs into a self-learning autonomic system which adapts over time taking into account business policies, constraints and feedback including behavioural feedback. A comprehensive investigation with the simulator is being carried out to establish the relevant metrics (stability, adaptability, agility, learning rates etc.) and the control levers (cycle times, thresholds, exploration policies etc.) leading to a design of what an "autonomics black-box" might look like.

3.2 Autonomics system design

In the following figure, a block diagram of the "autonomics black-box" is depicted.

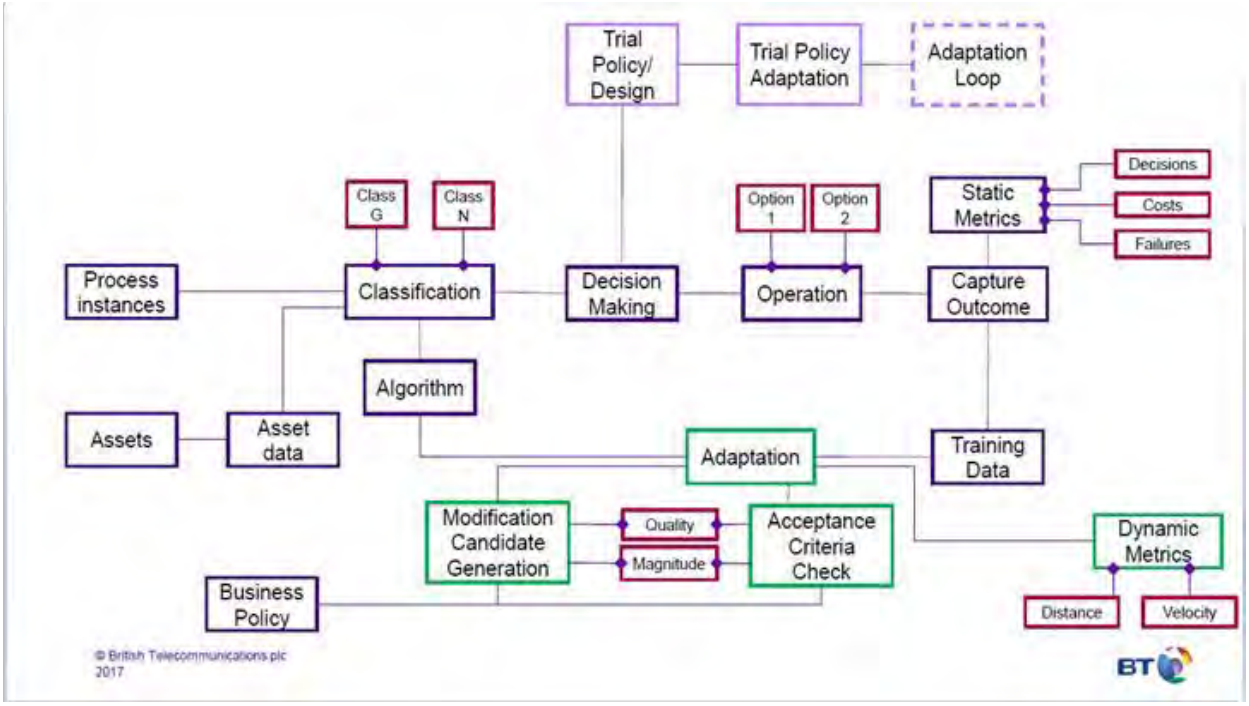


Figure 1: BT's "Autonomics black-box" design

In this figure, the control loops, the static and dynamic metrics, business policies and acceptance criteria, and how all these form a closed-loop around the system are depicted. This is unique and a novel way of dealing with the problem. First of all, in contrast with most of the machine learning systems, it is an active learning system where the predictions lead to decisions. These decisions do not lead to human driven actions but the action is automatic with the role of the humans in the loop being more of a monitoring/observing role. Furthermore, in comparison with open loop adaptive systems where the "AI-assisted analysis" leads to mapping the detected states to actions, it is offering a closed loop approach where static and dynamic requirements are continuously measured and the errors are corrected like in control systems. However, these corrections are connected to higher level business goals in double loop learning systems, managing in this way the business risks as well. We also propose a set of metrics to measure the quality of adaptation as well.

More detailed designs that include structural and behavioural descriptions are currently under development in order to model the system specification from several viewpoints.

4 V&V challenges for these systems: Properties Verification

Once such a "autonomic system blueprint" has been identified, applying this design to other applications is the next challenge, i.e. an implementation in software of an autonomic overlay on a business process, including the relevant interfaces and controls.

Software engineering verification and validation methods are absolutely paramount. in order to ensure the "correct" transition of autonomic algorithms to their applicability to business contexts and therefore contribute to their adoption and commercialization.

Beyond the traditional verification properties of safety, correctness, deadlock conditions, liveness that are used for system verification, adaptivity related properties are emerging from these autonomic applications.

In [5] the adaptivity properties were defined as follows:

- Stability, if the autonomic process will eventually converge to a stable and expected result.
- Accuracy, how close the resulting system is to the expected.
- Short settling time, how fast the system adapts and reaches the desired state.
- Small overshoot, not requiring unacceptable amount of computational resources for the adaptation.
- Robustness, operate within limits even in unforeseen conditions.
- Termination, the system operation is deadlock free for example.
- Consistency, same as ACID properties in transaction systems [6]
- Scalability, the system must be able to scale for increased demands of data and processing time.
- Security, the target system, data and components shared must be ensured for confidentiality, integrity and availability.

In the same paper, these properties were mapped to quality attributes of performance, dependability, safety and security.

The above are all equally important to be addressed. However, for the purposes of our industrial context and the adoption of the autonomics design blueprint to other business applications, emphasis on ***stability*** and ***robustness*** has been prioritised.

Last and not least, a new emerging property has been identified, the ***explainability*** property and its importance is considered paramount for all AI systems. The need for explainability of AI algorithms has been identified in the literature for some time now. However, it recently became even more important due to new data protection act rules (GDPR 2018) [7] and due to the requirements for wider applicability of AI to several application areas. BT's autonomics team has recognized this through several sources and identified that the explainability of AI algorithms is vital to ensure their adoption and commercialization. AI algorithms need to provide information for their decisions and operation at appropriate points in order to be able to be trusted and accountable. For example, in an autonomous cars crash, a court of law must be able to "trace" the AI decisions in order to identify causes and accountability. To the best of our knowledge, the explainability property is a new area and there isn't any research addressing this property.

5 Conclusions and Future work

Concluding, we can say that there is strong industrial need for advanced software and system engineering approaches and particularly for assuring properties and formal methods in the area of autonomic systems design and adoption. These highly dynamic and complex systems will require new methods to address their requirements.

Specifically, existing formal methods have not adequately addressed the above AI-related properties due to the complexity and unpredictability of the problem [8]. Techniques such as model-checking, probabilistic model-checking have been applied with the known problems of these methods such as state explosion and computational requirements. More needs to be developed in the forthcoming years to assure AI properties and ensure AI adoption.

Our future research plans include the description at system-level of BT's industrial autonomic blueprint system. Initial verification through simulation and testing will take place at the system level. Then, adoption of verification methods for adaptivity properties and development of new methods when required are the following steps.

6 References

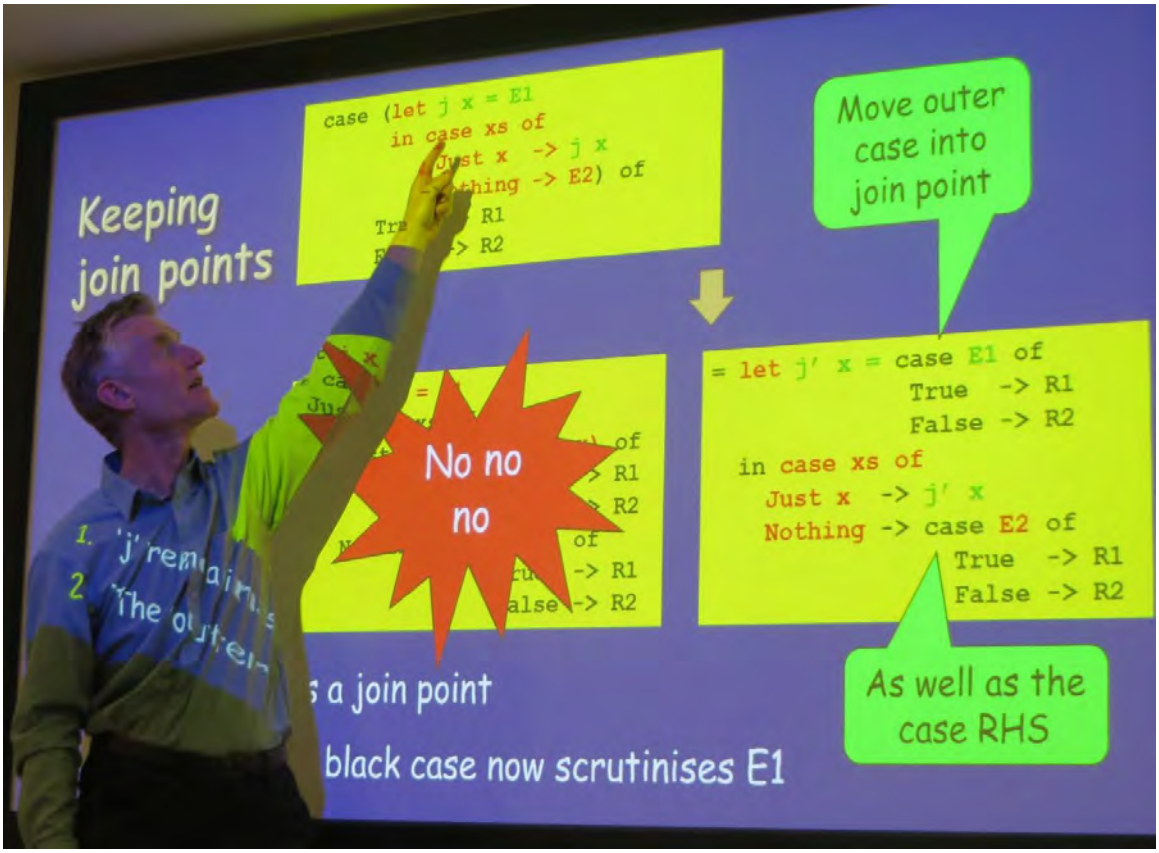
1. Dua, D. and Karra Taniskidou, E. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science. (2017).

2. F.D. Macías-Escrivá et al. / Self-adaptive systems: A survey of current approaches, research challenges and applications Expert Systems with Applications 40 (2013) 7267–7279
3. IBM Corporation: An architectural blueprint for autonomic computing. White Paper, 4th edn., IBM Corporation,
4. McCann J, Huebscher M, A survey of Autonomic Computing: degrees, models and applications, *ACM COMPUT SURV*, Vol: 40, ISSN: 0360-0300 (2007)
5. Norha M. Villegas, Hausi A. Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. 2011. A framework for evaluating quality-driven self-adaptive software systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS '11). ACM, New York, NY, USA, 80-89. DOI=http://dx.doi.org/10.1145/1988008.1988020
6. M. L´eger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in reflective component model. In Proceedings 13th International Symposium on Component Based Software Engineering, (CBSE'10), volume 6092 of LNCS, pages 74–92. Springer, 2010.
7. R. Pereira. Will GDPR hinder or harness the power of AI? (May 2018) Available online: https://www.itproportal.com/features/will-gdpr-hinder-or-harness-the-power-of-ai/
8. Calinescu, R., Kikuchi, S., & Kwiatkowska, M. (2012). Formal Methods for the Development and Verification of Autonomic IT Systems. In P. Cong-Vinh (Ed.), *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development, and Verification* (pp. 1-37). Hershey, PA: IGI Global. doi:10.4018/978-1-60960-845-3.ch001.

Photographs of speakers at FACS events

by Jonathan P. Bowen

Erika Abraham, University of Aachen, Germany, speaking on *Symbolic Computation Techniques in SMT Solving* at the London Mathematical Society on Thursday 2 November 2017.



Simon Peyton-Jones, Microsoft Cambridge, delivering the Annual Peter Landin Semantics Seminar on *Compiling without Continuations* at the BCS London office on Tuesday 6 December 2017.

Rob Hierons, Brunel University (right), introducing Mohammad Mousavi, University of Leicester (left), at the BCS London office on Thursday 22 March 2018.

Mohammad Mousavi speaking on *Model-Based Testing Cyber-Physical Systems: Theory and Practice*.

Rod Chapman, Protean Code Ltd, speaking on *The Fumble Programmer* at the BCS London office on Wednesday 25 April 2018. The audience includes (left to right) Richard Bornat, David Lightfoot, and Brian Wichmann.

Botond Virginas, British Telecom, and Sofia Meacham, Bournemouth University, speaking on *Autonomics and their verification from BT's Industrial Perspective* at the BCS London office on Monday 21 May 2018.

Report on FACS-LMS Seminar

Symbolic Computation Techniques in SMT Solving

Prof. Erika Abraham, University of Aachen

2 November 2017

Held at The London Mathematical Society, De Morgan House, London, WC1B 4HS

BCS-FACS and LMS, the London Mathematical Society, have a tradition of holding a joint seminar towards the end of each year on a subject of mutual interest. The 2017 seminar was on the subject of symbolic computation and the satisfiability of propositional formulae. Professor Abraham's abstract follows:

Abstract: 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 the 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.

Tim Denvir (*using material by Erika Abraham)*

Annual Peter Landin Semantics Seminar

Compiling without continuations

Professor Simon Peyton Jones, FRS (Microsoft Research)

Peter Landin (1930 - 2009) was a pioneer whose ideas underpin modern computing. In the 1950s and 1960s, Landin showed that programs could be defined in terms of mathematical functions, translated into functional expressions in the lambda calculus, and their meaning calculated with an abstract mathematical machine. Compiler writers and designers of modern-day programming languages alike owe much to Landin's pioneering work.

Each year, a leading figure in computer science will pay tribute to Landin's contribution to computing through a public seminar. This year, the seminar took place after the BCS FACS AGM on 12th December 2017 at the BCS Southampton Street, London HQ.

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 Simon will 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. Simon will 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.

Simon Peyton Jones spoke on the wonders of optimisation within the Glasgow Haskell Compiler, a sophisticated high-performance compiler for the Haskell functional language. The compiler broadly operates by performing a series of high-level transformations within the Core intermediate language to achieve high performance executable code. Simon noted that Landin also recognised the importance of control flow as well as data flow in his work, a point which is particularly relevant to this seminar.

The problem Simon outlines in his talk concerns the *avoidance* of the generation of a certain class of low-level code patterns as a part of the compilation process. Such a class, if not somehow handled and dealt with, would necessarily lead to potentially disastrous code *replication*. This replication would have a significant knock-on effect for control-flow optimisation, making that even more challenging to perform.

The elegant solution described by Simon is to soundly extend the Core intermediate language with a new language construct called a *join point* that permits the right kind of sharing to be introduced. The join point performs semantically just like a conventional "let" or "letrec", which automatically defines a localised intermediate function to abstract out the common code that would otherwise be replicated in further manipulations for optimisation.

Because of the locality and the way it would be used, an important observation is that join points are essentially "functions" that can always be implemented by the compiler as pure jumps (i.e. to control-flow labels), which thus avoids the expense and overheads of introducing explicit thunks/closures. For example, this implies that (mutually) recursive join points necessarily correspond to "tail" recursive calls.

With this new construct in use, it turns out that the modular design of the GHC meant that the various phases could be easily updated to accommodate and make use of join points – with the consequence that certain optimisations happen naturally as a part of code generation (i.e. simpler code).

Finally, it was really quite appropriate that this talk was given in honour of Peter Landin, one of the pioneers of mathematical operational semantics and the application of lambda calculus. As noted earlier, join points form a lower-level *control flow abstraction* that, because of the way they are used, they can equally be represented in terms of conventional jumps to control-flow labels. This strongly echoes the way that Landin was one of the first to introduce a control flow abstraction, the J operator, to represent control flow transfers – which, in turn, was an early precursor to the use of continuations in semantics and compiling technology (see discussion in [1]). And so it all comes full circle!

Simon provided us with a most enthralling deep-dive into the world of optimisation for functional languages. The talk concluded with some interesting questions and a drinks reception – which made for a convivial end to the evening.

Paper: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/join-points-pldi17.pdf
Slides: https://www.bcs.org/upload/pdf/compiling-without-continuations.pdf

Brian Monahan and Margaret West

References:

[1] Hayo Thielecke, *An Introduction to Landin's "A Generalization of Jumps and Labels"*, Higher-Order and Symbolic Computation, 11, 117–123 (1998), Kluwer

The Fumble Programmer

Roderick Chapman, University of York

Abstract This talk has a main aim to present the idea of the Fumble = Formal + Humble Programmer and how to combine Formal with the PSP in professional software development industries. It was presented by Rod Chapman, an independent consultant software engineer with years of personal experience in the area and international talks, and visiting professor of University of York.

Turing's lecture back in 1947 states that the programming should be done in such a way that frequently investigating identities (invariants for us) should be satisfied at all times if possible. Turing also stated that the machine interprets whatever it is told so communication to the machine has to be unambiguous.

Twenty-five years later, Dijkstra wrote "The Humble Programmer". In this book, he emphasizes the importance of avoiding bugs from the start and not fixing them afterwards. And he claims that the improvement in Quality that results is free as it prevents problems early on. The realisation that we should prevent problems from occurring and not build the software and then prove its correctness was formulated.

In order to write software that is verified by construction, the programmer should first of all realise how hard it is to achieve this and how bad the human mind is in tackling this complexity. Using verification tools at an earlier stage and having to be corrected at every step, makes you humble! And by being humble, you stand a chance to verify earlier and avoid the costly process of fixing afterwards.

This realisation is also an integral part of the Personal Software Process (PSP) where quality and cost are interrelated and where the defect-repair costs are highest in testing and during customer use than earlier in the design cycle.

Once the programmer is convinced to become ***humble***, the next natural step is to become formal. Why?

Thinking and Tooling exposes ambiguity, incompleteness, contradiction and semantic inconsistency.

Formal notations exhibit semantic consistency. They mean the same across all compilers, target machines, verification tools, the person that wrote it, the person that maintains it, etc.

Formal methods enable longevity and soundness.

A lively discussion on the soundness of verification tools took place with the speaker stating that the tool vendors with unsound tools are stating that soundness doesn't matter. The conclusion was that verification tools should be sound after all, in order to be trusted. A "social" proof for soundness was suggested by the speaker. This idea was based on feedback created and made public by industrial case studies formulating social evidence of success stories.

We concluded that we need the composite of a ***Formal + Humble = Fumble*** Programmer, and we discussed the current state of the art in formal notations with unambiguous semantics that provide hope towards that direction, such as SCADE, SPARK Ada, Eiffel, CakeML, Cryptol.

The talk concluded with a quote from Peter Amey, SPARK Team : ***Formality in verification makes you Humble..."It's like Jazz – hard at first, but worth it in the long run..."***

Sofia Meacham

A Report on the FACS Seminar

Model-Based Testing of Cyber-Physical Systems

Mohammad Mousavi (University of Leicester)

22nd March 2018

Abstract: Professor Mousavi talked about joint work spanning several years and which is ongoing. The team involved is spread across various Universities including Leicester, Pernambuco (Brazil), Halmstad (Sweden) and Eindhoven (The Netherlands).

By way of an introduction to the seminar Professor Mousavi dissected the title and explained the key words.

Model-Based: Build a model which abstracts from reality, simplifying the actual situation to a model which adequately reflects the major actions of the system and/or its environment. The simplification is easier to verify but it must be tested for conformance; checking that the model behaves in a way that is adequately close to 'reality'.

Testing: Generating test cases from requirements (as specified in a model), executing them on the system under test, comparing the outcomes with what is expected (by the model) and reaching a verdict about the quality of the system under test.

Cyber-Physical System: Incorporates control with communication and computation (the team involved in this project concentrate on control and computation). The main examples concern automotive systems. Most vehicles being produced today rely on vast numbers of microprocessor-controlled subsystems and millions of lines of software. The cost of the software is increasing both in absolute terms and relative to the overall cost/value of the entire vehicle. In the future we may well see most innovation taking place in the software, which will be updated via the cloud!

The behaviour of most physical world situations is governed by differential equations. Computer-based models that are used to mimic/control these are based on finite state machines and labeled transition models. To be of use these two, real and discrete, representations must be 'close'.

(Control Theory uses other mathematics which could perhaps be similarly approximated by computational models.)

Each model needs to be checked for conformance against its implementation. Technically, (tau, epsilon)-conformance is used to check the tolerance (epsilon)

in a numerical property of interest during a time interval (tau). We need to reject non-conforming systems so as to guarantee soundness. This is achieved by adjusting the sampling rate and/or the error margin in a suitable conformance analysis algorithm. Such a model-based testing scheme is outlined.

Moving from theory to implementation we need to determine/locate changes in the dynamic system so that we can calculate sensible error margins and adjust the sampling rate accordingly. This will probably require several iterations and from this we get an initial process flow diagram and we were shown such a generic process sketch.

We then move on to Case Studies, of which three were mentioned:

1) Engine Fuel Controller
2) Pneumatic Suspension System
3) Platooning - Running vehicles in close up convoys.

The first two were discussed fleetingly and illustrated by means of flow diagrams showing the main control functions and the factors that necessitate their change. The main illustrative example is Platooning and this was introduced by a video. The main idea here being that when a truck is running close behind another truck it suffers considerably less air drag and hence its fuel consumption is much lower. (As was pointed out by an American member of the audience, US truckers have been doing this for years - travelling at high speed, only a matter of inches apart. Perhaps these days communications can be a little more responsive than CB radios etc. This reminded me of 'Smokey and the Bandit'.)

So, we reach the goal of the seminar. We were presented with a collection of models, in diagrammatic form using Matlab, of system designs for the lead and follower vehicles in a platoon. Each pair used a different (more detailed) mode of communication starting with 'ideal' (instant or direct) connection. These were talked through in outline. Following from two of these models, the ideal one and one of the 'connected' ones, were looked at in some detail. Specific models of the various components/actors were given and graphs of the resulting position + velocity + acceleration relationships were presented. It really does look very promising.

But all is not yet done. The group is actively researching test case generation and have a process for adjusting parameters (in the right order) to guarantee soundness and have a prototype tool to support these phases.

This work now needs to be generalised so that the current models can be applied in other Cyber-Physical Systems (and use appropriate test data etc.) and demonstrated in more substantial case studies. And they welcome new partners in this endeavour.

Professor Mousavi was gracious enough to take questions throughout his presentation and there were some extra questions at the end of the talk. After the seminar there was an opportunity to 'network' (as they say). Professor Mousavi has recently become one of the editors-in-chief of the prestigious Elsevier journal 'Science of Computer Programming'. He has also agreed to join the FACS committee, to take charge of the 'testing' subgroup and to organise related seminars.

All in all, a most successful evening even though attendance was lower than expected.

John Cooke

FME Fellowship Award

FM 2018 International Symposium on Formal Methods

Sunday 15 July 2018

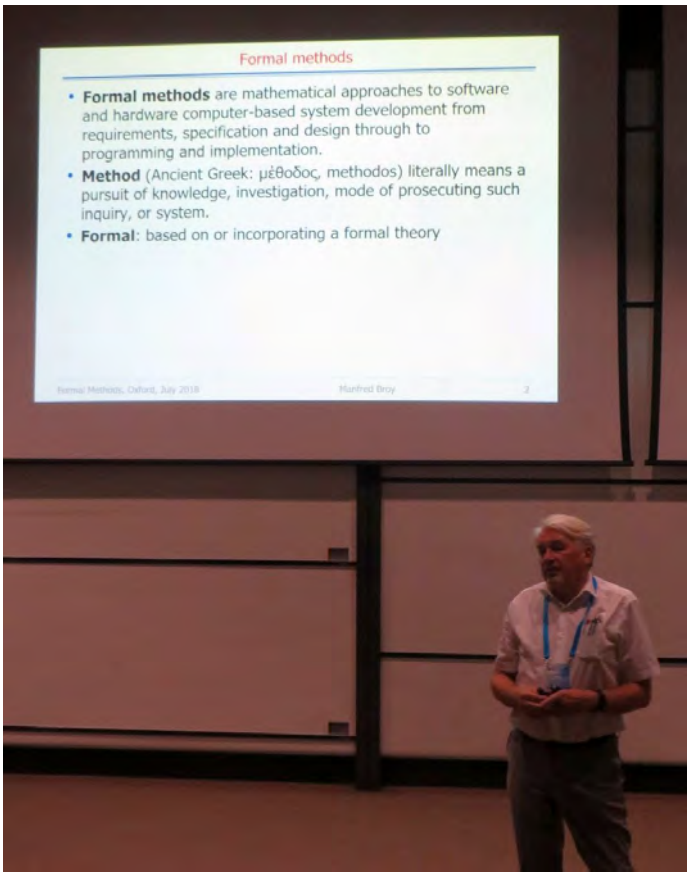
Held at the Mathematical Institute, University of Oxford

Ana Cavalcanti, The University of York, chair of Formal Methods Europe (FME) and a member of the BCS-FACS committee, introducing the FME Fellowship Award ceremony, sponsored by BCS-FACS.

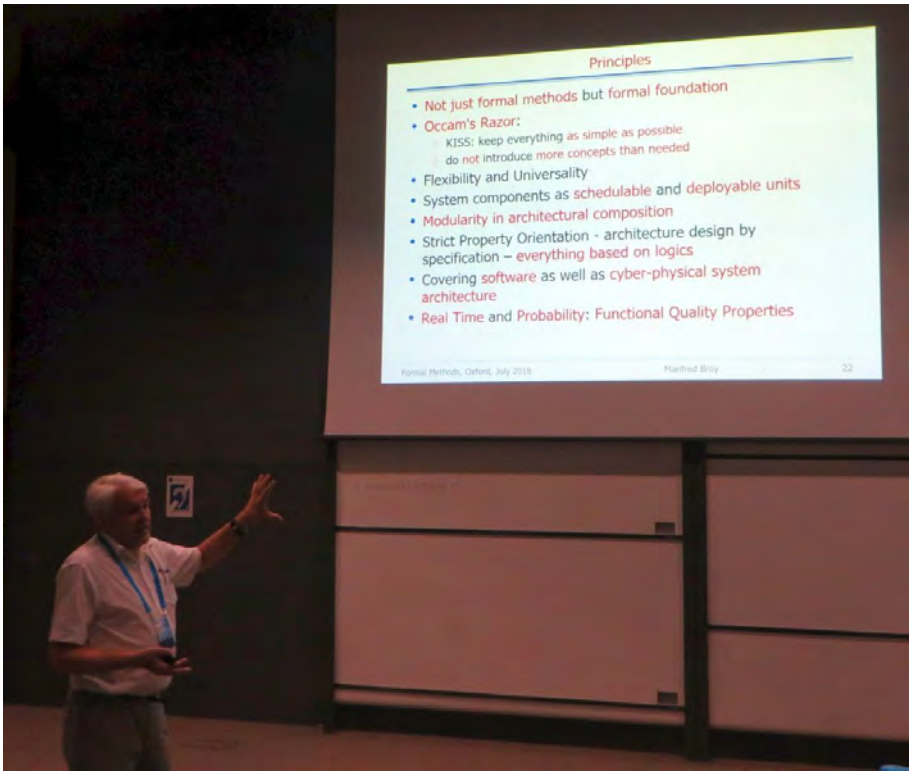
Manfred Broy, University of Munich, Germany, receiving the FME Fellowship Award with FME representatives at the *FM 2018 Symposium*.

Manfred Broy delivering his lecture on the *Formal Foundations of Software and Systems Engineering*, after receiving the FME Fellowship Award at the *FM 2018 Symposium*.

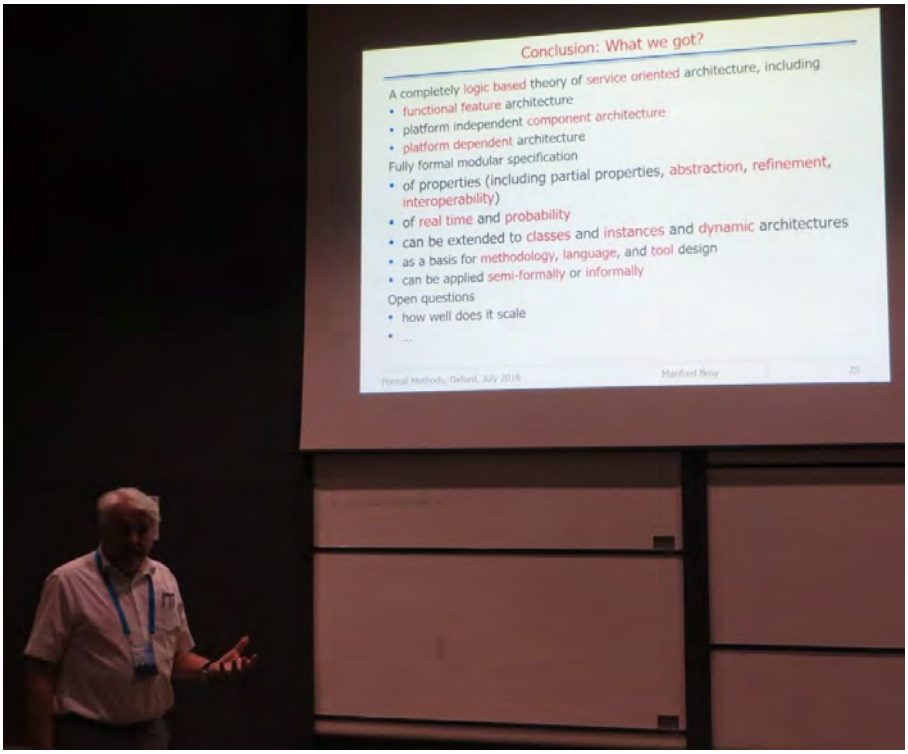
Manfred Broy introduces the formal foundations of software and systems engineering.



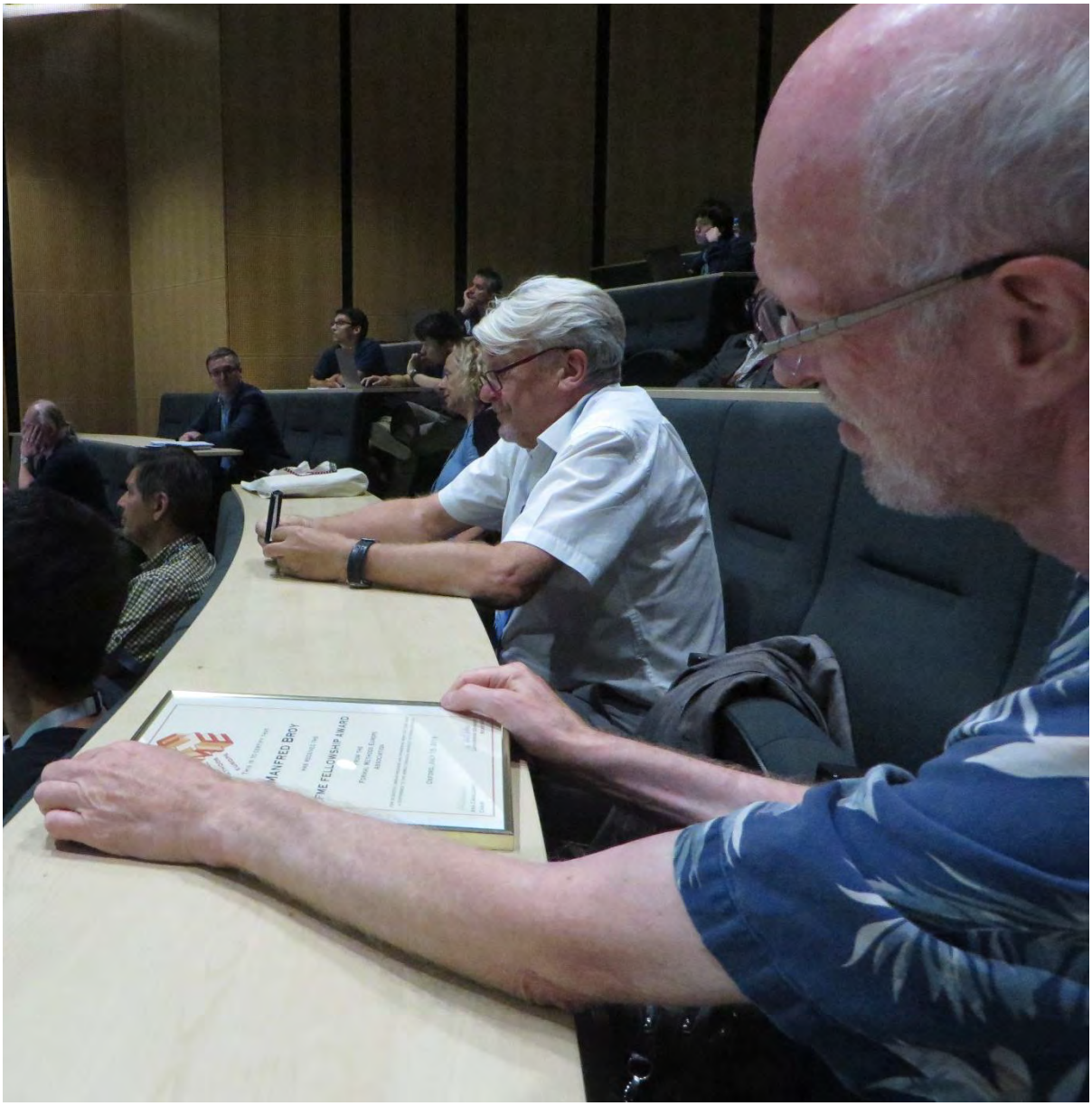
Manfred Broy presents the derivation of the term “formal methods”.



Manfred Broy presents his principles of formal foundations.



Manfred Broy presents his conclusion on the current situation.

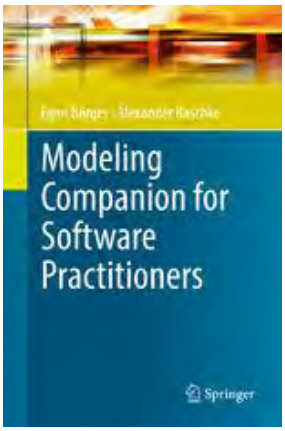


Eric (Rick) Hehner, University of Toronto, Canada (foreground), with Manfred Broy's FME Fellowship Award certificate and Manfred Broy (background), after receiving the FME Fellowship Award at the *FM 2018 Symposium*.

Jonathan Bowen

Book review

Börger, Egon and Raschke, Alexander, *Modeling Companion for Software Practitioners*. Springer, 2018. XXI+349 pages.
ISBN 978-3-662-56639-8. eISBN 978-3-662-56641-1.
DOI: 10.1007/978-3-662-56641-1



There are many formal methods potentially available for a software practitioner; so many that it can be difficult to evaluate which is best or appropriate for a specific software engineering project. In computer science academia, small examples are typically presented in papers promoting different formal methods. In practical application, scaling is a huge issue. If a technique does not scale, it will be of no interest in practice. Model checking is one approach that has gained popularity because of its relatively good tool support and the automation that it provides. This technique works well to a limited level, although this has increased in scale dramatically with Moore's Law helping to improve speed of execution, together with better algorithms and developments in software support. However, above a certain size, model checking is no longer feasible.

For larger software development projects, formal specification becomes more appropriate. This can aid in other ways such as allowing refinement to an executable program or for determining suitable software testing in a systemic manner. Model-based specification has proved particularly helpful and practical. Software engineers are used to considering a model for the system under development, even if done informally with natural language or diagrams. Using a formal approach, the state of the system can be modelled at an abstract level and operations on this state can be specified. VDM (Vienna Development Method) was an early example of this approach. The Z notation, although a general formal specification language, is nearly always used in a state-based manner, using a standard style for this. Later the similar B-Method was

proposed, designed for better tool-based support providing refinement all the way to executable program code.

ASM (Abstract State Machines) has also been proposed as a state-based approach to aid formal specification and refinement. The Workshop on Evolving Algebras (held in 1994 and 1996) became the ASM Workshop from 1997 to 2005. The original Z User Meeting (first held in 1988) became the ZB Conference (covering Z and B, first held in 2000). The ASM Workshop then merged with the ZB Conference to become the ABZ Conference (first held in 2008, covering ASM, B (and Event-B), Z, and later including other state-based formal specification approaches as well, such as Alloy, TLA, and VDM). This conference continues to this day, most recently in 2018.

A previous book entitled *Abstract State Machines: A Method for High-Level System Design and Analysis* by Egon Börger and Robert Stärk appeared in 2003 (also published by Springer) and introduced the ASM approach as both a textbook and handbook. The ASM notation is a form of pseudo-code based on finite state machines using abstract data structures. A "ground model" (so called because it should be grounded in reality) acts as a reference model for a design and stepwise refinement is possible towards a concrete implementation.

This current book under review resulted from two visits of the first author to the University of Ulm in Germany, the institution of the second author. It is aimed more explicitly for use in self-study by software practitioners, although it can also be used by students, and emphases the modelling aspects of ASM. Both authors are academics and in practice it may be that the latter use is more popular, but the first aim is laudable.

The book is divided into two parts on *Modeling* and *Implementation*. The first part with six chapters introduces modelling and refinement using ASM, including a variety of examples, covering concurrent systems, context awareness, business processes, and distributed systems. The second part includes three chapters on the syntax/semantics of ASMs, the CoreASM interpreter for executable ASM models using a restricted ASM language, and the graphical Control State Diagrams (CSD) approach with a graphical editor and conversion to ASM. An appendix includes some ASM models used in the rest of the book.

At the end of the book are a good set of references and a three-page index. The latter could be more comprehensive to improve use of the book as a reference work. The chapters are interspersed with a total of 65 exercises. Associated

online resources can be found under http://modelingbook.informatik.uni-ulm.de giving teaching material such as slides associated with chapters, answers to some of the exercises, and further examples (in PDF files and even ZIP file LaTeX sources). These additional resources (freely available under a Creative Commons license and still being developed further at the time of writing this review) make the book even more attractive for teaching on an advanced software engineering course at university final year undergraduate or masters level.

All successful formal methods have a community of practice built up around them and ASM, like B and Z, has such a community associated with it. Choosing between different formal methods can easily depend on which community a user becomes most affiliated. Learning to use a formal method well, especially writing a good formal specification, can take six months, although the ability to read a formal specification takes less. Once this time has been invested, another formal method must be significantly better in some way to make time spent learning it worthwhile. In industry, providing even a week for a training course to learn a new technique is considered a significant investment. This book does at least provide a new resource for practitioners that wish to invest the time learning ASM, whether alone, as part of an industrial course, or on an advanced university course. Producing such books is an important part of building a community around a technique.

The examples provided in this book are by necessity of limited size for didactic reasons. The question remains as to how well the approach scales and scaling is not explicitly mentioned in the book. The availability of tool support for ASM is of course to be welcomed and hopefully this will continue to improve. It would be pleasing if a future book in this potential trilogy could be co-authored with a genuine software practitioner, including the practical issues of scaling up for use in a selection of real industrial systems. Perhaps this co-author will be one that reads the current book under review and gains inspiration from it. The first two ASM books have been published 15 years apart (in 2003 and 2018). It is to be hoped that a third even more practical book on applying ASM in the large will be published in less than 15 years hence.

Prof. Jonathan P. Bowen, London South Bank University

Forthcoming events

Events Venue (unless otherwise specified):

BCS, The Chartered Institute for IT
The Davidson Building, 5 Southampton Street, London, WC2E 7HA

12 October	*Unifying Theories of Refinement*, He Jifeng, *Shanghai.*
17 October	*Coresets at the heart of Big Data*, Stephane Chretien, *NPL.*
1 November	*Verifying CSP and its offspring*, Bill Roscoe, *Oxford*. Joint event with the London Mathematical Society **Venue:** London Mathematical Society. De Morgan House, 57-58 Russell Square, London, WC1B 4HS
10 December	FACS AGM followed by: Landin Seminar, given this year by Don Sannella, *Edinburgh.*

Details of all forthcoming events can be found online here.

FACS Committee

Formal Aspects of Computing Science Specialist Group

Jonathan Bowen
FACS Chair; BCS Liaison

John Cooke
FACS Treasurer and Publications

Paul Boca
FACS Secretary

Roger Carsley
Minutes Secretary

Tim Denvir
Co-Editor, FACS FACTS

Brian Monahan
Co-Editor, FACS FACTS

Ana Cavalcanti
FME Liaison

Rob Hierons
LMS Liaison

Mike Hinchey
Lero Liaison

Keith Lines
Government and Standards Liason

Sofia Meacham
Seminar Organiser

Margaret West
Inclusion Officer and BCS Women Liaison

Eerke Boiten
Chair, Cyber Security Subgroup

John Derrick
Chair, Refinement Subgroup

Mohammed Mousavi
Chair, Testing Subgroup

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.

As well as the official BCS-FACS Specialist Group mailing list run by the BCS for FACS members, there are also two wider mailing lists on the Formal Aspects of Computer Science run by JISCmail. The main list <facs@jiscmail.ac.uk> can be used for relevant messages by any subscribers. An archive of messages is accessible under http://www.jiscmail.ac.uk/lists/facs.html, including facilities for subscribing and unsubscribing. The additional <facs-event@jiscmail.ac.uk> list is specifically for announcements of relevant events. Similarly, an archive of announcements is accessible under http://www.jiscmail.ac.uk/lists/facs-events.html with subscribe/unsubscribe options. BCS-FACS announcements are normally sent to these lists as appropriate, as well as the official BCS-FACS mailing list, to which BCS members can subscribe by officially joining FACS after logging onto the BCS website.