ISSN 1744-1986

Technical Report N° /

Jon G Hall and Lucia Rapanotti

,

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

20First Century ProblemSolving

Research Note: SMH#001

$e^{i\pi} + 1 = 0$ for Computing

Jon G. Hall and Lucia Rapanotti
The Open University, UK

Abstract Euler's identity relates, in just seven symbols, some of the most fundamental entities and operations in mathematics. It has been described as 'the most beautiful theorem in mathematics' and 'the greatest equation ever.'

If twenty-four hours is the clock of seven thousand years of mathematics, then Euler's identity is stated just one hour before midnight, computing (with Babbage's mechanical difference engine) arriving, fashionably, a few minutes later; Software Engineering crashed the party a mere 7 minutes ago.

Many have characterised elements of software, some have managed great beauty. Here we present another characterisation of the relationship between a software system, its context, the need it satisfies, its stakeholders and its validation, deduced from Rogers' definition of (general) engineering.

Introduction

It is said that Gauss used Euler's identity

$$e^{i\pi} + 1 = 0$$

as a benchmark for mathematical excellence:

> 'if this formula was not immediately apparent to a student on being told it, the student will never be a first-class mathematician.' (Derbyshire 2003, Page 210)

Euler's identity is very highly regarded in mathematics, for instance being described by the readers of Mathematical Intelligencer as the "most beautiful theorem in mathematics", Wells (1990)[1].

That there are mathematical things of beauty begs the question in computing: What constitutes computing beauty?

Certainly, there are many beautiful computing products; I'm using one now. But what about computing as engineering? Should we expect to have something like Euler's identity? Computing as engineering is a practical discipline without the support of hundreds of years of mathematics behind

1. Wells (1990) had suggested $e^{i\pi} = -1$, but this was indicated by one reader as less beautiful than the form of Euler's identity above.

it. Should we hold out any hope of finding a simply expressed relationship dense in meaning between its basic entities?

In this paper, we propose such a relationship. To approach it, we first show how we can capture engineering so as to be able to represent it in symbols.

Mr. Euler, meet Mr. Rogers

GFC Rogers, writing as recently as 1983, defines engineering as:

> 'the practice of organising the design and construction of any *artifice* which *transforms the physical world* around us *to meet* some *recognised need*' [*emphasis added*] (Rogers 1983)

As such, engineering is a form of problem solving.

Part of the utility (and beauty) of Rogers' definition is its vast, yet precise, applicability: as well as aeronautical, nautical and civil engineering, it can be seen to include financial, organisational, and educational engineering too, not to mention scientific engineering – the design of experiments and method, for instance – and, critical for us, computing. Under Rogers, computing as engineering becomes the practice of organising the design and construction of an artifice – an information system or the like – that meets a recognised need in a real-world environment.

Whereas this definition might be seen as sufficient in itself, in making formal Rogers' definition, we will be able not only to characterise the relationship between important parts or computing, but also that of computing to the other engineering disciplines too.

This formalisation we call Problem Oriented Engineering, or POE for short.

Problem Oriented Engineering

Suppose we let W represent Rogers' real-world environment, S his artifice, N his recognised need, and G the stake-holder whose need was recognised. An *engineering problem* (more simply, *problem*) is the *proposition* P_G

$$P_G : W(S) \ \textbf{meets}_G \ N \tag{1}$$

whose truth value indicates that, when S is installed in the environment W, their combination meets the need N to the satisfaction of stakeholder G. We would, of course, like to arrange that a problem is *solved with respect to the stake-holder* if and when the proposition is true.

From such propositions, we get a problem solving process in POE by relating problems one to another as relate the propositions that represent them. We don't go into too much detail here but, as an example, if both

Michael and *Gordon* are stake-holders in the same problem *P*, with *Michael* more critical as an engineer than *Gordon*, then we can say

$$P_{Michael} \Longrightarrow P_{Gordon}$$

or, if *Michael* says a problem is solved then *Gordon* will agree. *Gordon* and *Michael* are not *forced* to agree, of course, and there may be cases when *Gordon* is satisfied but *Michael* is not.

To organise the design and construction, we can take such implications and make them problem solving steps, writing, for instance[2]:

$$\frac{P_{Michael}}{P_{Gordon}} \qquad \textit{Michael} \text{ stricter than } \textit{Gordon}$$

to mean that

> 'as *Michael* is stricter than *Gordon*, to solve problem *P* for *Gordon* it is sufficient to solve it for *Michael*.'

In computing, the strictest *Michael* can be is to demand a formal proof that the problem is solved, but such proofs are, in general, difficult to come by: in POE, it is not our intention to produce proof as such. The reader familiar with the propositional calculus may recognise the Natural Deductive style of our system and so be surprised that we do not have proof as our goal, our relaxation however allows us to guard problem solving steps with *justification obligations* the discharge of which establishes the 'soundness' of a problem solving step. One such we have already seen in "*Michael* stricter than *Gordon*."

Although such guarding adds an extra layer of complexity to transformations, we feel its benefits greatly outweigh this; they include that we can make problem solving stake-holder specific in the sense of arguing the *adequacy* of a solution with respect to a particular stake-holder, and not only its absolute correctness. It follows that stake-holders that are convinced by software testing can be convinced by problem solving steps whose justification is testing-based, whereas the much more difficult formal proof of correctness can be left as justification for those who require it. Such adequacy arguments capture the notion of *fitness-for-purpose* with respect to the stake-holder(s), which is an important tenet of engineering. A well-designed problem solving step in POE will preserve fitness-for-purpose.

What sort of needs are there?

Within computing, recognised needs are, ahem, recognised as consisting of functional and non-functional, or quality, parts. Functional needs are

2. In fact, this example is more than a design step, it is a design strategy in that it relates all such problems!

named for mathematical functions which express relationships between 'inputs' and 'outputs'. A simple example of a functional need is

'should add input x to input y to produce output z'

Qualities, on the other hand, talk of a solution's characteristics *in situ*, such as its reliability, its usability, even its colour, *etc.* As such, qualities can only be examined in a context: as a extreme example, the quality

'The system interface must be easily learnable by end-users'

will even be satisfied by a command line interface should the context of the system be a LINUX-based developer.

Functional needs are the Platonic ideals of recognised need and have a position in computing, and in particular computing science, reflecting this. Notwithstanding the achievement of formalists in codifying the machine and in producing techniques to reason about absolute correctness, in the real-world functional requirements are, simply, unsatisfiable.

The reason is that the real-world is a very aggressive place.

Schroeder et al. (2009), for instance, describe the corrupting effect of cosmic rays on memory chips in Google's server farms. In Google, as you might imagine, reliable software is mission critical. For use in Google's world, then, even formally proven-correct software must be hardened against cosmic rays, and the other real-world aggressors, within its environment, some of which – including humans – effect error, caprice and maliciousness, even!

This all boils down to the fact that, for a purely functional need F, the problem

$$W(S) \textbf{ meets } F \tag{2}$$

has no real-world solution S; even a proven correct solution can fail.

What's needed is a sort of get-out-of-*f*ail-free[3] card. Pick up the pink card for one I prepared earlier as an example of a quality requirement. As opposed to a functional requirement, you'll notice it comments on populations of behaviours – here, the failure level allowed in $1,000,000$ computations – rather than single ones.

Associating with functional requirements one (or more) qualities can cover the fallibility that arises through real-world-embedding. In comparison with Equation 2, if Q is an appropriately chosen quality, such as that

3. The board game Monopoly includes a 'get-out-of-*j*ail-free' card of use to incarcerated players.

above, then, in combination with F our pay-off is to be able to write:

$$W(S) \textbf{ meets}_{G} Q(F) \tag{3}$$

in which $Q(F)$ should be read as 'Expect F, subject to Q'.

But is this any more soluble than the original? Hold that thought...

You're late, Mr. Alexander!

A quality like Q is applicable very many different F and in many different contexts W, so Equation 3 can be seen as a recurring problem.

Treating it as such brings us to the Alexander et al. (1977) ideas on *patterns*:

> 'A pattern is a careful description of a perennial solution to a recurring problem within [a context], describing one of the configurations which brings life to [that context].'
>
> (Alexander et al. 1977)

Actually, our ellipsis was originally the phrase 'a building' (appropriate given that Mr. Alexander's was a definition from architecture). But. like Rogers' earlier definition, through a small change we can consider *computing* patterns[4].

The 'Recurring Reliability Problem' as we might now call it, as found in Google's server farms, is ameliorated by memory chips which employ Error Correcting Codes, or ECCs, at the chip level.

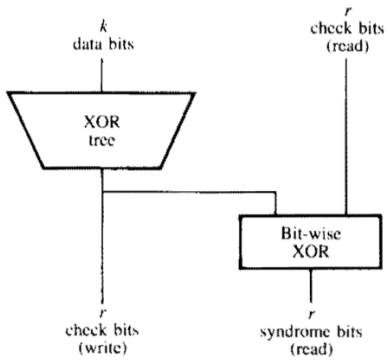

Figure 1: A recurring solution to the 'Recurring Reliability Problem': Error Correcting Codes. k data bits are combined and compared with r previously calculated check bits to detect memory failure. Source: Chen and Hsiao (1984)

The ECC perennial solution offers some protection to an ideal solution S in the context W. More generally, in the real world, an engineer would

4. Software patterns are, of course, well known; here we simply wish to emphasise that both hardware and software and whole socio-technical systems – can be the perennials solutions.

look for a 'perennial solution' corresponding to a particular quality Q. Supposing one exists, let's call it A_Q[5]. Using it to buffer S from W we arrive at:

$$W\left(A_Q(S)\right) \textbf{ meets}_G\ Q(F) \tag{4}$$

Of course, even with A_Q as protection, W may be still be too aggressive an environment for S. Whence...

And then there were four

In his 1986 paper, *And No Philosophers' Stone, Either*, Wladyslaw M. Turski observes that:

> There are two fundamental difficulties involved in dealing with non-formal domains (also known as "the real world").
>
> 1 Properties they enjoy are not necessarily expressible in any single linguistic system.
>
> 2 The notion of mathematical (logical) proof does not apply to them.
>
> (Turski (1986))

and continues with:

> '[...] experimental validation [...] is a necessary step in application software construction, and that at the same time it is the least conclusive and most influential stage of the process [leading to] software design paradigms which isolate the experimental steps from formal ones.' (*ibid*)

Essentially, because the real-world is aggressive and not amenable to formalisation, we should be careful to validate through experiments the properties of any solution: no matter what the formalist can prove *in vitro*, we should always validate *in vivo*[6] In effect, we should design a validation scheme, V, that varies with F, G, Q and W, with which to justify the problem solving step:

$$\frac{S \textbf{ meets}_G F}{W(A_Q(S)) \textbf{ meets}_G Q(F)}\ V \tag{5}$$

That's it.

5. *A* is for *A*rchitecture.
6. As Donald Knuth 1977 wrote to Peter van Emde Boas:

   "Beware of bugs in the above code; I have only proved it correct, not tried it."

Outroduction

Stepping back, this is a relationship between just nine symbols which – we argue – characterises computing as engineering. This problem solving step relates an engineering problem via a quality, a solution structure and a validation condition to a problem amenable to formalisation. Unlike Euler's equation, the symbols aren't constants. Like Euler's equation some of the most indicative parts of computing as engineering are there.

In his conclusions, not all of which are positive, Turski is suggestive of:

> *software design paradigms which isolate the experimental steps from formal ones*

Perhaps he hoped that there would be a single theory to bind both together; then again, perhaps not. Even so, it may be that the relationship we have forged in POE between functional and non-functional requirements, between structure and validation can provide a vehicle for the investigation of such paradigms.

In closing, it may be that, given that there is no *a priori* dependence for POE on computing, the engineering of software can, by expansion of the language of the context, the need and the solution, be melded with its sister engineering disciplines.

If so, perhaps computing is reaching its age of maturity!

Acknowledgements

Heartfelt thanks to Chris Tiernan, Managing Partner at Grosvenor Consultancy Services, for his insightful comments on earlier versions.

References

Problem Oriented Engineering Home. URL http://www.solvemehappy.com/. Last accessed: November 24, 2011.

C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language: towns, buildings, construction*, volume 2. Oxford University Press, USA, 1977.

C. Chen and M. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM J. Res. Dev.*, 28(2):124–134, 1984.

J. Derbyshire. *Prime obsession: Bernhard Riemann and the greatest unsolved problem in mathematics*. Joseph Henry Pr, 2003.

Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Memo to Peter van Emde Boas, March 1977. http://www-cs-faculty.stanford.edu/~knuth/faq.html Last accessed: November 21, 2011.

G. F. C. Rogers. *The Nature of Engineering: A Philosophy of Technology*. Palgrave Macmillan, 1983.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. doi: http://doi.acm.org/10.1145/1555349.1555372.

Wladyslaw M. Turski. And no philosophers' stone, either. *Information Processing*, 86, 1986.

David Wells. Are these the most beautiful? *The Mathematical Intelligencer*, 12(3):37–41, 1990. URL http://www.springerlink.com/content/71u5410642258635. doi:10.1007/BF03024015.