Journal of Universal Computer Science, vol. 13, no. 5 (2007), 593-601
submitted: 7/5/07, accepted: 25/5/07, appeared: 28/5/07 © J.UCS

Formal Methods
Guest Editorial

Richard Banach
(School of Computer Science, University of Manchester
Manchester M13 9PL, UK
banach@cs.man.ac.uk)

Abstract: This introductory paper gives some historical background to the emergence of formal methods, overviews what subsequently happened, and surveys prospects for the future. Brief introductions to the remaining papers in the Special Issue are given.

Key Words: Formal Methods
Category: D.2.4, F.3.1, B.2

1 Introduction

We'd better start by defining terms, since the phrase *Formal Methods* (FM) can have a wide range of connotations in Computer Science. For the purpose of this Special Issue, 'Formal Methods' means the use of mathematical techniques for the construction of computing systems, with a particular focus on the construction of industrial scale systems.

The focus on large systems is significant. The nature of the mathematics which describes computing systems is such that many appealing ideas that can work on tiny examples, simply become infeasible as the problem size gets large — or to put it in slightly more technical language, the *complexity* of many techniques grows far too fast for comfort as the problem size increases. One of the biggest challenges in the subject, is to get the maximum possible leverage out of techniques that are intrinsically limited by such complexity considerations. *Unfortunately*, it is the case that almost all techniques that offer an appealingly deep insight into a system definition suffer from this limitation to a greater or lesser extent. *Fortunately*, it is also the case that quite reasonable progress has been made in taming the complexity ogre in various ways, so that systems or subsystems of a useful size can be tackled.

This is not to say that the complexity ogre has been vanquished. Not at all — he continues to lurk, barely out of sight, just beyond the reach of even the best techniques. But it is the case that over the last ten years particularly, many specific gaps in his defences have been discovered, and specialised techniques and tools have been developed, that can wound him severely. Thus while the systems that can be handled by many of the rigorous methods that are around today, still typically remain a good order of magnitude or two smaller than the biggest systems that routinely get developed, they are nevertheless big enough to capture serious system functionality. The careful use of FM in the development of appropriately selected systems or parts of systems, contributes to the increasing levels of dependability that systems so developed nowadays exhibit.

2 A Little History

How did FM get going? At the beginning, in the twenties and thirties, the time of Turing and Church, before there were any actual computers, computation was accepted as a branch of mathematics connected with formal logic. In those days, programming was something done only in an 'in principle' manner, to show that some task could indeed be accomplished theoretically by some technique. Big systems were certainly not on anyone's mind.

A bit later, in the forties and fifties, real computers started to appear. Then, the interest was in keeping them working, with their thousands of valves which frequently burnt out. In those days, programming was carried out by a few skilled individuals, working on isolated tasks. Programs were necessarily small, since nothing existed on which to run big ones.

Slowly things began to change. Solid state devices began to replace valves, dramatically improving reliability, and increasing hardware speeds and memory capacity. The increased size meant that people started to contemplate and build bigger and more complex systems. With the advent of 'high level' languages such as Cobol and FORTRAN, software construction became more than something that just a few specialists did.

Projects to build big systems began to run into trouble. They completed late, over budget; often the final product didn't work. Many projects got into such a bad state that they got cancelled. War stories of the time remain entertaining to this day (eg. the operating systems team that met their deadline by delivering a blank tape, secure in the knowledge that the hardware was in no fit state to run *anything* — the hardware team commissioned the tape for a fortnight before they discovered the truth, by which time the operating systems people were in a position to deliver something sensible). It was the Wild West of system construction. Regarding the software side, the *software crisis* had well and truly arrived.

The point of this historical note is that the software crisis, a term coined at the 1968 NATO Software Engineering Conference, crept up on people; no one had been expecting it.

It was the complexity ogre at work. While systems remained small, keeping their various parts consistent during the development was a modest task. However, as systems grew in size, understanding them, and keeping their parts functioning and co-operating properly, grew much faster than the size of the system suggested. So the task of creating a large system that behaved as intended, soon became considerably bigger than people first presumed.

3 Responses to the Software Crisis

According to the 1968 NATO conference, the answer to the software crisis lay in changing system construction from an ad hoc skill, to an *engineering* discipline. The vision of those early days looked forward to system construction being done with the same rigour and predictability as was common in well established engineering fields.

In particular, it was recognised that mathematics would have to play an appropriate role in software engineering, just as it does in traditional engineering. In a sense, this was how FM came into being — one can see the emergence of the earliest techniques as having their roots in the research of that time.

What would a properly engineered system development technique be focused on? One can see the successful construction of a large system as being founded on three things.

- One needs to know what is required. Trying to build a system without a clear awareness of the requirements is like starting to build a bridge, and only several months after work has commenced, deciding where it should go. The nature of the real world means that 'perfect knowledge of the requirements at the outset' is often an unattainable ideal. However, one should try to attain it as far as possible, and to try to see to it that late changes in requirements refer to aspects of the system built later rather than earlier.

- One needs techniques that will turn the requirements into a working system in a dependable and traceable way. This dependability must apply as much to the dependable following through of requirements changes part way through the process, as it does to the implementation of the original requirements. The more rigour that can be brought to bear on this aspect, the greater the confidence in the resulting system.

- One needs to manage the processes above in a sensible way, so that what the people involved may eventually produce is (ideally) greater than the sum of its parts.

In the light of the above, one would have thought that system builders would have been keen to adopt rigorous, mathematically based methods, and that by now these would be commonly found within system construction practice. However this is far from the case. Why? One can posit several explanations.

- After the early days, most people involved in system construction no longer had a mathematically rich background. This created a natural reluctance to get involved with mathematically based methods, a trait which remains to this day.

- After the 1968 NATO conference, in contrast to its expectations and in tandem perhaps with the previous point, much emphasis in software engineering was put on the *management* and *process* aspects of system construction, to the detriment of incisive concern with the *artifact* being created. This is still largely true today.

- There is a *perception* that dealing with mathematical techniques is hard, and therefore a *conclusion* that introducing rigour into the system construction process is to be avoided. While there is limited truth in the perception, the introduction of increasingly sophisticated automation is making a big difference to the validity of the conclusion these days.

• There was a period in which FM, were, in the delightful phrase of Barroca and McDermid "both over-sold and under-used." The over-selling stemmed from the uncritical promotion of individual techniques without proper regard for the requirements context; the under-use was the inevitable consequence of the resulting bad press.

So, for reasons like those just given, formal techniques did not become mainsteam. What happened to software construction despite this? Well, things have undeniably got better. The use of increasingly high level structuring techniques, languages, APIs and tools, has certainly reduced very significantly the scope for projects to founder because of 'static shallow defects.' One might call a defect static and shallow, if it is relatively easy to describe, and is detectable by looking at the system description, expending work roughly in proportion to system size — i.e. it is the kind of thing one could build into a compilation system. Conservatism is also a great help, since it is a lot easier to succeed in building a system that is not unlike a system that others have already built, than to understand how to build something completely new.

But conservatively scoped systems, and static shallow defects are not the only players on the field. Semantically deep bugs can destroy the usefulness of a system. Most to be feared however, is uncontrolled and explosive requirements change. To this day, projects that are both novel and huge, can and do disintegrate because the requirements arena is too poorly understood at the outset.

Formal techniques, with their much greater semantic depth, offer the possibility of keeping these phenomena under control, at least in principle. The caveat is needed because the complexity ogre is never far away. Arbitrary semantically deep bugs can be prohibitively expensive to discover for large systems, as are the consequences of large collections of relatively arbitrary requirements.

4 The Hardware Scene

We've said a lot about software. What about hardware? Was there and/or is there a corresponding hardware crisis? A number of things have been helpful for hardware over recent decades. Unlike the pure 'bits' of software, hardware is a manufactured product. The manufacturing costs are very visible, the production timescales allow little room for manoeuvre, and over time, the capabilities of the underlying primitives have changed little in principle, even if the numbers parameterising these capabilities have changed significantly. Beyond that, the sheer cost of coping with a genuine design error (as exemplified by the famous Pentium^{TM} division bug) has fostered a culture of design reuse that the software field can only envy at present. And the Pentium^{TM} bug itself accelerated the much more rapid adoption of formal verification technologies in hardware compared with software. The resulting innate conservatism in the hardware field has helped to subdue the worst effects of the complexity ogre in recent years.

But there is no room for complacency. Moore's Law has delivered an exponential increase for transistors/chip over the last few decades. Alongside this, human design capability has also grown exponentially (aided for example by improving design automation). But the time constant for the latter is longer than that for the former. This has resulted in a steadily increasing 'design gap' between the potential functionality of a chip and the effort needed to design that chip from scratch. The culture of reuse of successful designs, not only hedges against errors, but helps to narrow the design gap where it can be applied. Besides this, the prospect of the 10 billion transistor chip brings with it a host of novel challenges involving the interaction of nano scale physics and manufacturing imperfection, which also impacts design. All of this generates a growing design crisis, qualitatively distinguishable from the software one, against which current techniques will not prevail.

5 The Formal Methods Landscape and Prospects

Thus we arrive at the situation today. History means that FM did not become the technique of choice in the software system construction mainstream — it is still primarily confined to critical developments in such fields as avionics, air traffic control, railway switching, medical electronics. However the advances of recent years have changed the prospects for wider adoption significantly.

We can group formal techniques into three broad classes, recognising that in fact, these days, ingenuity targeted at battling the complexity ogre increasingly combines ideas from any or all of them.

Model Based Techniques. Model based methods are the oldest in the FM armoury. They are based on building models of the state space of the desired system, and specifying the allowed transitions between states: permissible sequences of transitions are then defined implicitly. Models can be constructed at various levels of abstraction, and related by suitable refinements. At the lowest level, the models become executable, and the refinement(s) that relate them to the higher level models prove that they are correct with respect to those higher models. The complexity of the proof task can be formidable, but these days, by breaking the task down into small pieces, automation can accomplish a remarkable amount.

Behaviour Based Techniques. Some time after the introduction of model based techniques, temporal logic became an alternative way of specifying properties of systems. Instead of concentrating on states, sequences of transitions are the main focus. The advent of model checking brought automation to this field and quickly ran into the state explosion problem, in which the size of the state space that needs to be explored to verify a property makes the checking job infeasible. Much inventiveness has been brought to bear on this conundrum in order to extract impressively useful results from initially unpromising starting points.

Static Analysis Based Techniques. While some degree of static analysis has always been present, for example in the type checkers of even early compilers, more recently, the technique has been extended to be a powerful lever for inferring system properties. System definitions are approximated in various ways, and the approximations can be analysed to deduce properties of the original system. Of course something is lost in this process, and the results can give false positives for example. The challenge is to design approximations that minimise such phenomena, and to do so automatically.

So what of the prospects for the future? Well we can certainly be sure of one thing: the complexity ogre will not go away. People will always find it easier to build systems that 'mostly work' on a scale that will defeat a good proportion of state of the art rigorous techniques, since it is easier to build something which one *thinks* is right than to advance convincing arguments that it actually *is* right.

Another point is that most people will continue to not like mathematics, prolonging a reluctance to adopt mathematically based techniques in the mainstream. In many engineering fields, mathematical competence is *demanded* of practitioners. This did not arise by chance, but happened, unfortunately, as a result of disasters that mobilised the political will to *enforce* best practice standards. Will something similar happen with digital systems?

The prospects for uptake of formal techniques can be improved by breaking down the barriers between techniques. Small differences of emphasis can lead to unfortunate incompatibility. In this sense, the simple semantic picture at the heart of model checking has already accomplished a lot, but much more remains to be done.

Finally, the FM community can do more to stress the financial angle. **It is folklore in the community that projects undertaken using FM wisely, complete on schedule and within budget.** Quiet but persistent appeal to the bottom line instincts of commercial systems developers may yet do at least as much for the adoption of FM techniques as anything else.

6 A Quick Look at the Papers

If the preceding paragraphs have conveyed the feeling that the formal arena is a struggle between the deep insights that formal techniques can potentially deliver, and the depradations of the complexity ogre — and that despite this, valuable progress has been and will continue to be made, then they have succeeded. The papers in this collection highlight representative areas of activity in the vigorously developing FM field.

Our collection starts with a contribution from Sriram Rajamani of Microsoft Research, at Bangalore. With colleagues at Microsoft, Sriram was instrumental in the creation of contemporary tools used by Microsoft to verify protocol adherence aspects of device drivers in Windows Vista^{TM}. Remarkably, and almost uniquely in the formal methods landscape, these tools are able to routinely process tens of millions of lines of code. As such, they represent a very specific but very deep wound in the complexity ogre. Sriram's paper looks at software construction in the wide sense, surveys the state of the art in static checkers, and looks forward to when software might be created in a way in which all the high level aspects of the design remained solidly linked to low level code throughout the whole of the development process.

The next paper is by Connie Heitmeyer, who heads the Software Engineering Section in the Center for High Assurance Computer Systems at the Naval Research Laboratory in Washington, DC. Her paper, which describes the specification, validation, and verification of requirements, focuses on techniques for accomplishing these tasks with the SCR toolset of which she is the chief designer. SCR is based on Parnas's tabular techniques and has been successfully used in a number of real-world projects. The focus on requirements reflects the fact that requirements are now universally recognised as a key issue in the delivery of high quality systems.

Requirements are also stressed in the next paper — and though handled very different there, the common objective is clear: to wed an account comprehensible to domain experts, to one in a formal language suitable for further development.

Jean-Raymond Abrial's paper, next, illustrates the heights that the original 'proving programs correct' paradigm has reached these days. Based on theory that dates from the early days of formal methods, Jean-Raymond's B-Method has been in development for two decades or so, and this has resulted in the latest version, called Event-B, which the paper overviews. Event-B has refocused the theory on the most used and simplest parts of the earlier method, making it more accessible. Most importantly, from its inception, the B-Method has been developed hand in hand with thoroughgoing tool support. It is this latter aspect that has been responsible for the outstanding industrial successes that the B-Method has enjoyed, and which Jean-Raymond describes.

Since the famous Pentium^{TM} division bug, hardware vendors have taken a keen interest in formal verification techniques, in hopes of avoiding a repetition. John Harrison of Intel is in the thick of this activity on behalf of that particular manufacturer, and contributes the next paper about the role of formal verification in contemporary chip design. The implementation of arithmetical and mathematical functions in hardware provides a unique kind of challenge —which John describes— since their abstract definitions are so far removed from the bitwise manipulations at the chip level. Mastering the situation requires the integrated use of an unusually wide range of techniques.

Ed Clarke is one of the founding fathers of model checking. In the next paper with Flavio Lerda, these two authors give a review of model checking, from the basics of the original algorithms to contemporary developments. Judicious use of abstraction is vital if model checking is not to founder on all but the tiniest of systems, due to the ravages of the state explosion problem. Ed and Flavio's paper conducts the reader through the various techniques that the ingenuity of researchers has created to try to keep the state explosion problem at bay, including symbolic model checking, BDDs, the developments of model checking applied to software, infinite state systems, and ultimately, applications to hybrid and control systems.

The next paper is by John Rushby, who is Program Director for Formal Methods and Dependable Systems at the Stanford Research Institute. FM activity at SRI has always been heavily focused on tools, and SRI has produced many leading-edge FM tools over the years. The paper surveys some of the technical history of FM, complementing the more context oriented remarks in this editorial. This leads into a broad survey of extended static checking and a discussion of SMT solvers. SMT solvers, which combine the power of decision procedures for specific mathematical theories with general propositional reasoning are adding great power to automatic FM tools these days. The paper discusses industrial applications and looks forward to much more adoption in the mainstream.

The following paper is by Jim Woodcock and myself. Jim has for many years been at the interface of academic formal methods and its applications in industrial contexts. The paper is about the Verification Grand Challenge. This is a long term project that Jim leads and whose goal is to bring about the needed confluence and commonality of purpose that will be instrumental if formal techniques are to take up a prominent position in the mainstream system development methodologies of the future. The paper overviews the long term strategy of the Grand Challenge, and describes early achievements; these have mainly been concerned with the full mechanisation of the Mondex Electronic Purse, a ten year old formal development, aspects of which I was revisiting in the year or two prior to the commencement of the Grand Challenge.

Our final paper is from Anthony Hall, whose professional career has been concerned with delivering systems of the highest possible dependability. As a systems construction practitioner who has to satisfy his customers' demands, Anthony looks at formal methods with the gaze of someone who cannot afford the luxury of changing the problem to neatly fit the technique of interest, a temptation to which we academics are at times prone to yield. So there is great encouragement in his positive take on the possibilities offered by formal methods today, and no better way to round off this survey of the current state of the art than his closing remark that "**it is demonstrably possible to succeed with formal methods now.**"

About the Guest Editor

Richard Banach is a Senior Lecturer in the School of Computer Science at the University of Manchester U.K. He has a B.Sc. in Mathematical Physics from the University of Birmingham U.K., and a Ph.D. in Theoretical Physics from the the University of Manchester.

In the early 1980s he worked for International Computers Ltd. on the 3900 series mainframes, at the hardware/software interface. In 1986 he joined the Computer Science Dept. at Manchester University. His main interest then was the hardware/software interface of the Flagship Project, for which he formulated the MONSTR graph transformation technique, Flagship's 'machine language.' This subsequently developed into an interest in graph rewriting, and semantic techniques in general.

Since the mid 1990s, his main research interest has been formal methods. This led in particular to the development of the retrenchment technique. Retrenchment is intended to help bridge the gap between the absolute precision and brittleness of logical descriptions and their verification technologies, and the complexity and inevitable imprecision of the wider requirements milieu. Retrenchment, and its interplay with other verification techniques continues to be his main interest. He has published over 60 journal and conference papers.