Professional Opinion on Skills with Formal Description Languages

compiled by Peter Bernard Ladkin and Bernd Sieker
Draft Version 1, 2012-02-25

The document relates a e-mail discussion which recently took place in February 2014 on the System Safety mailing list, a list of some 200-300 engineers and technicians worldwide with an interest in system safety, maintained by the Faculty of Technology of the University of Bielefeld, Germany. It is lightly redacted, and includes only opinions directly relevant to the initial questions as formulated below. Some observations on the discussion follow the contributions.

The compilers of this document refer to themselves throughout as "author" and "authors", to avoid possible confusion with other meanings of "compiler".

Request

by Peter Bernard Ladkin on 20140215, http://www.systemsafetylist.org/0885.htm

> *I would like to gather opinions on the necessity of some skills. I would argue that programming dependable systems, and dependable-software engineering in general, requires some facility with formal description languages (FDLs). FDLs are necessary for requirements specification and analysis, and they are also necessary for the refinement steps that occur between requirements and code, as well as for any kind of static analysis of code. They are necessary for validating compilers. Classical proposition and predicate logics count amongst the most common and basic FDLs; a skill in expressing some kinds of assertions in predicate logic has often been regarded as necessary for a basic informatics education. I would think some understanding of how FDLs work is essential for any work in dependable engineering of SW. Do people agree? If so, could you please give me some evidence?*

Selected Responses

Nick Tudor, Aeronautique Associates, [private communication]
http://aeronautique-associates.com/aeronautique_about_us.htm
Nick has worked for the Royal Air Force and Qinetiq and is a member of the RTCA DO178 committee Subgroup 6 (Formal Methods).

> *The use of logic, discrete maths, upon which is built formal methods, for the development of digital systems and software is .... entirely ......logical. I think it highly important that logic is taught to undergrads and that it should be compulsory. The reason we have systems and software that continually fail to live up to expectations of customers can at least in part be attributed to poor software engineering practice let alone poor programming.*

John Knight, Professor of Computer Science at the University of Virginia, is author of the book Fundamentals of Dependable Computing for Software Engineers (C&H/CRC 2012).
http://www.cs.virginia.edu/~jck/ http://www.systemsafetylist.org/0887.htm

> *.... I teach the discrete mathematics course to second-year students at the University of Virginia. The course is required for computer science and computer engineering majors.*
>
> *In my opinion, the situation is as follows:*
>
> - *To a very large extent, all software is critical in some way. Even gaming software,*

the failure of which could lead to loss of reputation, market, income, etc.

- *All of engineering rests to some extent on mathematics.*
- *Discrete mathematics (the term I would use where you have used logic) is the mathematics of computer engineering.*
- *We will not make progress against the serious assurance challenges we face unless we apply mathematics.*
- *All computer engineers should be trained to understand, appreciate and apply discrete mathematics.*

How does one integrate discrete mathematics into the undergraduate curricula? Here is the way I do it:

- *I define a problem that the students can easily understand first, and then I show how some aspect of discrete mathematics can solve it.*
- *A major problem I use is specification. I use specification to motivate sets, propositions, predicates, relations, functions, etc. For each topic I state bits of specification as examples.*
- *I bring all this together by introducing a declarative language based on discrete mathematics, Z, and show the students both how easy and how effective such languages are.*
- *At various points I note that such formal techniques will help the students make money (because formal techniques are valuable) and help keep them out of court (because formal techniques are the state of the art).*

....

I am talking about software products that are part of engineered computer systems which will subject others (possibly the general public) to risk. Higher education has a responsibility to prepare professional engineers to perform that engineering. That education needs to make it clear that:

- *Engineers are responsible for what they do.*
- *Engineering is a profession not some amateur activity.*
- *Mathematics is an essential component of professional computer engineering.*

In response to the comment from Les Chambers:

"We must find a way to bring formal methods out of the lab and into general use."

I generally agree. But I note that we have industrial strength systems such as SPARK Ada, industrial scope use of such systems such as the NATS iFACTS system, and substantial evidence from Peter Amey and his colleagues that applying such technology is cheaper and better than the informal alternatives.

Steve Tockey, industrial software developer at Construx http://www.systemsafetylist.org/0890.htm and http://www.systemsafetylist.org/0900.htm :

First, include me in the camp with PBL and Dr. Knight: formality in software is critical, even if the application being developed isn't. As a "software professional" for over 37 year I'm constantly amazed at how informal 99% of the software development community is and their utter blindness to how much trouble that very informality leads to. I often say, "We are an industry of highly paid amateurs".

In my experience, not only is it possible to take a more formal approach to software development, it's at least an order of magnitude easier if we do.

Second, from a practical perspective most developers--me included--tend to have a hard time with all of the "upside-down As and the backwards Es" usually associated with truly formal approaches like Z, VDM, Larch, etc. There's a very interesting and relevant paper, IMHO:

Jeannette M. Wing, A Specifier's Introduction to Formal Methods, Computer, September 1990

To me at least, the gist of Wing's paper is that I should be able to use a comfortable surface notation, such as UML Statecharts and Class Diagrams as long as I have a clear mapping to a defined, formal underpinning (such as could be expressed in Z, VDM, etc.). I don't want to deal with upside-down As and backwards Es, and neither does my customer. But if I can have a formally define-able ****single**** *interpretation of something like a UML Statechart then I get the all of the benefits the underlying formality provides together with the comfort, familiarity, and ease-of-use of something like a Statechart.*

While we're on the topic of Statecharts, I would also like to toss out this fascinating (to me, anyway) paper:

M. von der Beek. A Comparision of Statechart Variants. In W.-P. de Roever, H. Langmaack and J. Vytopil, editors, Formal Techniques in Real-Time and Fault-Tolerant Systems, number 863 in Lecture Notes in Computer Science, pages 128148. Springer Verlag, September 1994.

If I remember correctly .... the author describes about 17 significantly different valid interpretations of themsemantics of Statecharts. In other words, von der Beek shows how the same Statechart has the potential to have as many as 17 different valid interpretations. Each interpretation would lead to a developer producing very different (behaviorally) code as an implementation of their specific interpretation--which is probably different than the interpretation intended by the creator of the Statechart. Talk about safety implications...

So while a lot of people are comfortable using simple modeling notations like Statecharts and Class Diagrams, to be really useful those diagrams have to be backed up by a single interpretation as defined by some underlying formal semantic. It doesn't necessarily have to be the same interpretation used by everyone else, but everyone here has to know—and agree--which single interpretation is in use.

This is how I interpret PBL's use of the term "FDL" in his original posting, some convenient-to-use surface notation that's backed up by a single, solid, formally defined interpretation.

Third, while "model-based development" may be a newish buzzword to a lot of people, some of us practitioners have actually been doing it for almost 30 years. And getting tremendous benefit out of doing it, too, I might add. For the life of me I simply can't understand why more software development isn't done that way. The majority of the problems commonly encountered on typical software projects simply goes away with model-based development. I'm in the process of putting together a position paper explaining what "model-based development" means to me and why I think it's vitally important for the software industry to move in that direction as quickly as possible. I'll be happy to share a draft of that paper with *anyone who wants, just send me an e-mail asking for the "What is Code?" paper and I'll give you the draft when it's polished enough to share. I should have it share-able within a couple of days.*

Finally, getting back to PBL's questions:

"I would think some understanding of how FDLs work is essential for any work in dependable engineering of SW. Do people agree? If so, could you please give me some evidence?"--I agree 100%. Evidence? My position paper on model-based development, von der Beek's paper on variability in Statechart semantics, and Wing's paper. I also have at least two case-study write ups where use of (well, at least semi-formal) FDLs made a huge difference in the project's outcome.

(paraphrased) "What specific formal logic needs to be taught?"--I'm not sure yet of the specifics. But I am sure that it has to be enough to provide the formal foundation for the modeling notations we find useful in software projects. I can tell you what modeling notations/methods I find useful, we can work together to figure out what underlying formal foundations would be needed.

[Further contribution]

.........a new version of the "Guide to the Software Engineering Body of Knowledge" (aka "SWEBOK Guide", in this case SWEBOK Guide V3) has recently been released by the IEEE Computer Society. You can get your own PDF copy by going to http://www.swebok.org and then clicking on the "PDF (free)" link under "Get the SWEBOK Guide". This new version is a significant update from the previous 2003 version. Among other things, several new "Knowledge Areas" have been added.

A quick survey shows the following sections talking about formal methods of one sort or another:
Chapter 1: Software Requirements
  Section 1.4.5: Formal Analysis
  Possibly Section 1.6.3: Model Validation

Chapter 9: Software Engineering Models and Methods
  Section 9.1.4: Preconditions, postconditions, and invariants
  Possibly Section 9.3: Analysis of models
  Section 9.4.2: Formal Methods

Chapter 14: Mathematical Foundations

So my point is that the SWEBOK Guide people have at least recognized the need for formality in professional software development. Hopefully, as industry and academia adopt SWEBOK Guide as a description/definition of "what software engineering is" then there will necessarily be an increased emphasis on formality.

Les Chambers, system safety and software safety contractor,
http://www.systemsafetylist.org/0892.htm :

I am in furious agreement with Steve Tockey. If we are to deal effectivelywith the large code bodies that are now the norm. We must find a way to bring formal methods out of the lab *and into general use. The primary reason for this is that the larger a code body becomes the less likely it is that anyone but the author will ever review the code. Ask your formal methods detractors, "do you really want to sit on an aircraft with avionics software that has been eyeballed by only one person - the author?" I have done enough test work to be fully conversant with the ugliness that some programmers are capable of. Believe me, you do not want to be anywhere near a life critical software intensive system infused by same. Of course I am assuming here that formal methods support automated static and possibly dynamic code review. Am I correct? I know from personal experience that manual review of very complex designs and code is substantially simplified by semiformal methods such as state machines.*

The mission of any engineering discipline is to reduce science to practice to solve real world problems. Unfortunately the profession seems to be stuck at semiformal methods. The current impediment is money; finding investors who want to pay for the software engineer to formally specify program before it's written. On one missile program I worked on it was hard enough to get the cash to keep the mission function flow diagrams up-to-date, let alone specify anything with formal methods. Simplification and maybe tool support is the solution. I hope that's possible - over to you. If we could abstract this problem in terms of human commercial behaviour, industry experience with state engines is a good case study. Vendors of programmable logic controllers took a commercial decision to emulate relay racks described by ladder diagrams when they first released their products. This allowed existing electricians to use the new tools with minimal training. They could have followed the other option of logical decomposition of all control problems into cooperating state engines. That would have required retraining of the entire cohort of electrical tradesmen and, as such, would probably have been commercial suicide. Of late, some products do support the state engine concept though the last one I came across was a really ugly implementation.

In the period 1975 to 1985 I worked in an environment where all control systems were implemented with cooperating state engines. We simplified the concept so plant operators could easily understand what was going on. We never did use the word State. We used step. We didn't use the phrase state engine. We used the term sequence control unit. Operators understood that. That's how operator manuals were organised. After a few years we even had operators - with no high school education, who had run a state based machinery and learnt the concepts by doing - writing control programs. Is this possible with formal methods? This is a righteous question that I would like to see answered by the enormous cohort of brainpower corralled in the world's universities. A solution would be hugely beneficial to the industry at large. My humble suggestion for a starting point is to start with something that programmers know and move on from there. I am passionate about this subject because, in a previous life, I was the poor electrician responsible for figuring out how a room full of relay racks was controlling a high-speed lift (elevator). The massive complexity of these control systems required 2 to 3 years practical experience before you became useful. It was a craft with high priests who "had the knowledge" and walked on water. Much like current practitioners of formal methods. That does not have to be. State engines simplify. State engines support analysis of design. I hope the same will be true one day for upsidedown As and backward Es. In the meantime, the joint ACM, IEEE task force , who recently released their latest curriculum guidelines for undergraduate degree programmes in computer science are very supportive of teaching formal methods in their software engineering modules. See:
http://www.acm.org/education/CS2013-final-report.pdf

Bertrand Ricque, Program Manager at Sagem, Chair of the French standardisation committee for functional safety, member of the IEC Maintenance Team for the software parts of IEC 61508, http://www.systemsafetylist.org/0893.htm :

> *My vision is (as usual) even more pessimistic. What you write is not pertinent only for computer sciences but also for system engineering. We have guys here coming from top ranking engineering school who would never pass your logic test.*
>
> *I am currently involved in a relatively critical system design. It happens that most engineers don't understand that some kind of logical reasoning is necessary to switch from a given specification to a more refined one. It seems incredible to them that a refined specification (call it model) would not end automatically in something satisfying the higher level requirements. I have to use very specific, primary school level, analogies to have them discover the point ! And this is at Master + 2 years level.*
>
> (In response to John Knight):
>
> *"Engineers are responsible for what they do." (JK)*
>
> *This depends on the countries and their local engineering cultures and legal system. In France engineers are not personally responsible. The boss of the company is responsible. Engineer is not a regulated position such as dentist or lawyer ...*

Michael Tempest, critical-software developer, http://www.systemsafetylist.org/0894.htm :

> *I observed two patterns over the 15-odd years that I worked for an airborne military systems house (where I wrote mostly 178B level C"commercial software", since the general practice was to design systems where software could not contribute to hazards):*
>
> *1) The programmers with an electronics engineering background found it very difficult to express an argument (for example, why it is okay to violate* ****this**** *coding standard rule* ****here*****) and even more difficult to review an argument. I fall into this category and I learned to do it badly (by the standards of this list).*
>
> *2) The programmers with a computer science or computer engineering background generally understood formal methods and were often enthusiastic about them, yet were not able to apply them cost-effectively. Curiously (to me), they were not much better at expressing arguments. They did do slightly better at reviewing arguments.*

Philip Koopman, Professor of Computer Science, Carnegie-Mellon University, embedded-system specialist, http://www.systemsafetylist.org/0898.htm :

> *I'm all for software that actually works going into products via whatever approaches are effective (mathematics, peer reviews, etc. -- they can all play an important role). While I haven't spent a lot of time trying to get formal methods adopted, I have spent a lot of time trying to get organizations to do peer reviews/inspections as a baby step toward crawling out of the muck of chaotic software development practices.*
>
> *What I've found is that they often just can't bring themselves to put emphasis on an activity that is not directly contributing to the specify=>implement=>test path. (Sometimes they*

even skip "specify" so they can get right down to the business of writing buggy code, but that's another story.) I can speculate that the higher level managers assign value to creating code and testing activities. They assign essentially no value to defect prevention. These higher level managers seldom have training in software engineering (or even computer science/engineering).

From what I've seen our students act the same way. It is all about getting the code written, "tested," and slipped past the grading gatekeeper, however messy that process is. Essentially no thought or value is placed on avoiding defects in the first place. This approach appears to have been trained into them in intro programming courses.

I run a senior/MS course that pushes students through a project that is difficult to survive unless you practice bug prevention. Most of them get the message and are running effective peer reviews by the end of the course. I fancy that by the time they've completed the course and learned the lessons, they'd be ready to adopt more formal practices (but not before -- they are skeptical even of peer reviews for several weeks). I touch briefly on formal methods, but the math is more than I can squeeze into my course on top of everything else I need to cover. Perhaps if this sort of experience happened early in their education instead of at the end it would help motivate them to learn and practice the right mathematical skills, and they'd be eager to take a course on that topic.

But to effect change, IMHO first we have to convince our non-software-engineering/non-safety critical colleagues that this is something worth doing. I've never had much success doing that. Part of it is probably that as researchers we mostly specialize in throw-away non-critical code. It's tough to convince someone that teaching a topic is important if they've never found it important themselves.

A software-based critical-systems engineer with a large European engineering company (private communication to an author):

Within [my company], most direct use of Informatics in System Design is linked either to SCADE [the development environment from Esterel Technologies which uses the specification language Lustre] or to Statecharts (in some form or other). Therefore I would agree with Steve Tockey's assessment.

Application of MBSE (including at least somewhat formal specification) started in [my company] systematically in the late '80's and 1990's and in the .... domain, [in which] I work, started in [R&D] in early '90's and properly for [product] specification ..... in 2001. The ..... specification relates to the design phase before going into [SW development], and the application of [the applicable standard]......

In .... we still only do formal proof in the odd [R&D] project, but do rely on people's ability to properly interpret Statecharts. This can include some quite involved logic in the transition expressions, and sometimes includes non-trivial issues of Statechart-to-Statechart communication.

Our suppliers use UML, we tend to use Simulink block decomposition, which means that the system engineers also need to understand the way a block diagram manages signal dependency. Unfortunately, the message-based communication semantics of UML are rather different from this, and it does cause confusion. I'm often a little disappointed with people's understanding of these issues even when dealing with engineers with 30 years' experience.

For what it's worth, my education (Computer Science, preceded by a part-completed degree in electrical engineering) covered logic (by example through Prolog), Boolean logic (with Karnaugh maps etc.) included as part of Maths module 2, although in my case, this was a repeat of what we had covered in digital electronics) and Functional Programming (using Haskell or an earlier language Miranda). Only the Functional Programming bit was elective.

My personal opinion is that Statecharts should be covered as a compulsory part of almost any Informatics course. They are as useful for GUI design as they are for embedded logic. It would be lovely to get students to the point where they could use some kind of proof tool to check properties of 2 or more communicating statecharts. Obviously, you can't do Statecharts without understanding basic set theory and some amount of temporal logic.

On a related topic, I am appalled by the inability of most programmers to understand concurrency, the use of semaphores and other mutual exclusion mechanisms, etc. I wasn't taught this at university either. I had to learn the hard way in my first job. I still have a stack of papers on all sorts, including Rate-Monotonic analysis, that took me so much effort to collect and learn, I won't discard them! This is even more important now that most CPU's are multi-core.

Martyn Thomas, founder of Praxis (now Altran), a pioneer in the use of FDL and unambiguous semantics of programs with the SPARK toolset, editor with Prof. Daniel Jackson and Lynette Millett of the US National Academy of Engineering study Software for Dependable Systems: Sufficient Evidence? (NAP, 2007) http://www.nap.edu/catalog.php?record_id=11923 , http://www.systemsafetylist.org/0908.htm :

It's good to see the late, great Peter Amey's name appear in this thread. Here are some of his papers:

http://www.macs.hw.ac.uk/~air/rmse/c_by_c_better_cheaper.pdf
http://www.altran.co.uk/fileadmin/medias/0.commons/documents/Whitepapers/Logic_versus_magic.pdf
http://www.macs.hw.ac.uk/~air/rmse/Industrial_strength.pdf
http://www.bowdoin.edu/~allen/courses/cs260/readings/amey.pdf

The Tokeneer project for the NSA is also essential evidence:

http://www.adacore.com/sparkpro/tokeneer

Dewi Daniels, Founder, Verocel and associate of Aeronautique Associates, assessor of the C130J software for conformance to UK MoD SW standards, http://www.systemsafetylist.org/0914.htm :

I graduated with a degree in Computing Science from Imperial College, London in 1981. I don't think I could have wished for a better foundation for my career in computing. The teaching emphasised general principles that have stood me in good stead throughout my career instead of specific technologies (which have long since been rendered obsolete -- we used IBM 029 card punches in my first year).

Yes, we were taught logic, computing theory and formal methods. I remember being taught set theory, predicate logic, Turing machines, the theory of algorithms, logic programming

(Imperial was big on Prolog in those days) and program proof (the text book used was Dijkstra's seminal "A Discipline of Programming"). I was delighted to be able to put these skills into practice when I joined Program Validation Limited in 1990 to conduct program proof of a family of Full Authority Digital Engine Controllers and to help develop the SPARK Examiner. While I agree with other posters that formal methods deserve much wider application, the main point I want to make is that I have found my education in formal methods to be invaluable even on projects that make no use of formal methods. Throughout my career, I have been surprised by the exceedingly poor quality of requirements produced by many organisations. In particular, I have come across many very smart and experienced programmers who seem totally unable to write detailed and precise requirements without resorting to writing pseudo-code. I believe that my experience of writing pre- and post-conditions has greatly improved my ability to write clear and concise English-language requirements that do not constrain the implementation. Furthermore, I believe the main benefit of
my undergraduate education was the ability to approach problem-solving in an analytical and methodical manner and to express my ideas clearly and concisely, which are skills that I believe would have benefited me greatly in any career I had chosen to follow.

The course at Imperial also emphasised the principles of good software design and the importance of abstraction. I owe a great debt to my programming tutor, Iain Stinson, whose ideas on software design have influenced me throughout my career. He taught the principles expounded by Dijkstra, Wirth and Parnas. I find the main difference between average programmers and great programmers (not that I consider myself one of the latter) is the latter's ability to find the right abstraction, eliminating unnecessary complexity. The best programs look so simple that they give the misleading impression of looking as if they were easy to write.

The course at Imperial was very much a software engineering course rather than a computer science course. Imperial was rather unusual at that time in teaching project management as part of the undergraduate curriculum. The teaching included practical programming projects where we were given the opportunity to lead a project team. The main textbook used was Brooks' "The Mythical Man-month". I'm quite depressed how many managers still think you just divide effort by team size to get project duration.

Unfortunately, statistics was taught at Imperial by the mathematics department, whose lecturers considered us mere engineers to be inferior to their mathematics students, and who had an excessively theoretical approach to the subject. Fortunately, I had studied 'A' level statistics at grammar school under an excellent teacher called Mr Roberts, who instilled in me an understanding of both the power and the limitations of statistical methods. I am always amazed by the way that statistics are misused by people who should know better.

The tutors at Imperial were right to emphasise general principles over specific technologies. We were taught to program in Pascal (mostly), Fortran, Cobol and Simula 67, none of which are in common use today, yet the principles I was taught are as valid today as they were in 1981.

Derek M. Jones, a software developer, http://www.systemsafetylist.org/0920.htm :

Formal logic is all well and good for small systems but it does not scale. I think you should explain this important issue to your students.

A mathematicians point of view: "Highly complex proofs and implications of such proofs" *http://rsta.royalsocietypublishing.org/content/363/1835/2401*

The practical usefulness of formal logic for anything but the smallest problem is wildly overblown in computer science and I continue to be amazed by the claims made by the proponents of this approach: *http://shape-of-code.coding-guidelines.com/2013/03/10/verified-compilers-and-soap-powder-advertising/*

Thomas replied to Jones, http://www.systemsafetylist.org/0926.htm :

I have seen mathematically formal methods used successfully on industrial projects involving more than a hundred engineers and thousands of person-days of effort. I have seen formal proofs carried out on safety-critical metro systems by industrial engineers at Siemens Transportation and on the message choreographies for electronic commerce systems by software engineers at SAP.

In my opinion, the larger and more complex a system is, the more it requires the use of abstraction to master the complexity; abstraction without formal logic is just arm-waving.

Jones replied, http://www.systemsafetylist.org/0928.htm :

It is also arm-waving to say that formal methods will scale to large (i.e., more than a few hundred lines) systems.

Thomas suggested that Jones defines what he considers to be "large scale" and then try to find an example of "large-scale use" of formal methods which fits the definition http://www.systemsafetylist.org/0930.htm .

Daniels addressed the above comment of Jones directly, http://www.systemsafetylist.org/0934.htm :

I am astonished by this statement. I was personally involved in the program proof of several thousand lines of code in the early 1990s, including the Full Authority Digital Engine Controller for the Rolls-Royce RB 211-535. Since then, there have been many much larger applications of formal methods, including SHOLIS, Tokeneer and iFACTS. Furthermore, there has been a noticeable reduction in the number of Blue Screens Of Death in Microsoft Windows in the large decade; I understand this has, to a large extent, been achieved by the adoption by Microsoft of a tool based on formal methods to detect defects in third-party device drivers (see e.g. http://research.microsoft.com/pubs/70038/tr-2004-08.pdf). I find your suggestion that formal methods have not been used on programs of more than a few hundred lines to be very surprising.

Jones's final assessment is, http://www.systemsafetylist.org/0936.htm :

I think a cost/benefit analysis of formal methods could well come out in favor of their use for very critical code, such as the software controlling the safety rods in a nuclear power plant.

[In response to Thomas's query as to what he considers constitutes "large-scale"]

It is possible to write large quantities of 'code' in some formal method language. The ISO

> *Modula-2 Standard is 600+ pages long and almost all formal notation. But as far as I know the only 'formal' tool they used involved the layout of the text.*
>
> *If you define formal methods as writing the formal 'code' and proving something useful with it, then..........*
>
> ....
>
> *I don't have any references to cost vs length of formal proofs; compute resources (i.e., time) vs length of formal proof is also another issue for which data is very sparse and vague.*
>
> *I know of cases where a few hundred lines have been analysed using formal methods to a degree that looks very solid.*
>
> *Thousands of lines? The NICTA work ......is in this ball park, but they assumed to much and I was not convinced.*
>
> *The analysis of the Viper processor springs to mind as an example of something non-trivial. I know there was lots of debate about that worked had actually proved and what constitutes correctness......*
>
> *I know of projects that started off using formal methods and found them too expensive for the benefits provided and also found they made it difficult to substantially change the software later (because the need to redo the formal side would have been prohibitive).*
>
> *So my view is that formal methods may scale to a few thousand lines, provided the code is not likely to substantially evolve over time (or the money is available for major reworking of the formal side).*

The Viper processor-verification effort mentioned by Jones, and the accompanying controversy, is described in Donald MacKenzie, Mechanising Proofs: Computing, Risk and Trust, MIT Press, 2001.

Heath Raftery notes, http://www.systemsafetylist.org/0940.htm :

> *The first paragraph in the Introduction of [the Microsoft paper referenced by Dewi Daniels] is particularly telling:*
>
> *"[Originally] the ultimate goal of formal methods [...] was to [...] rigorously prove programs fully correct. While this goal remains largely unrealized, many researchers now focus on the less ambitious but still important goal of stating partial specifications of program behavior."*
>
> *I hope I'm not misrepresenting Derek's argument, but it seems to me this is the core issue. Derek claims that using formal methods on a small slice of a larger project does not constitute an example of formal methods scaling to a large project. I'm inclined to agree with Derek's contention that formal methods scale very poorly to large projects, and this paragraph from the paper backs that up.*
>
> *I however, still remain hopeful, and look forward to reading more examples of large scale use of formal methods.*

Martyn Thomas suggests this misses the point, http://www.systemsafetylist.org/0941.htm :

> *.... Mostly, what you want to be able to do is to assure particular properties of programs. It's rarely enough to show that the program fully and correctly implements a functional specification, as this leaves open all sorts of safety and security vulnerabilities if (for example) the program encounters input outside the specified range. (If you disagree with this statement, please send me a functional specification for a real application that is sufficiently complete and unambiguous to exclude such issues).*
>
> *The tools that microsoft use to model-check the behaviour of device drivers have made a great difference to the reliability of Windows. I don't think this could have been achieved any other way - years of human reviewing and testing had proved inadequate to the task.*
>
> *So let me turn the point round. What would consititute "large-scale use of formal methods"? and why is the putative lack of such examples important? Are the "formal methods deniers" really only building systems for which the smaller scale use of formality would not bring the sort of benefits that Microsoft, Siemens, Airbus, Boeing, Lockheed Martin and Praxis have experienced?*

Michael J. Pont, a former academic who has founded a critical-embedded-systems company, SafeTTy Systems, says http://www.systemsafetylist.org/0949.htm :

> *My present organisation is usually contacted by companies when they are having difficulty meeting safety / reliability requirements and / or when they are about to develop a product in compliance with IEC 61508, ISO 26262, DO-178, etc (perhaps for the first time). The companies that I deal with produce systems ranging from aerospace systems to household appliances.*
>
> *In many cases (probably the majority of cases that I see), there is limited evidence of "process" or documentation available from the "embedded team" when I arrive at the company door. Requirements documents are often* ****very**** *basic, if they exist at all. In many cases, this situation is inevitable because the teams are very small (even in large organisations), and the people involved have far too much to do. Sometimes the main contribution that I can make is to provide a fresh take on the problems, an extra pair of hands - and support for a case to "the management" for additional resources.*
>
> *My point with all of this is that - even if they were able to recruit some smart new graduates with lots of experience in formal methods - this would be unlikely to help the majority of the organisations that I see. Before these organisations are going to be in a position to make use of formal techniques, they need some more basic foundations (detailed requirements documents, linked design documents, code reviews, documented test procedures linked to requirements, etc).*
>
> *[I accept that it can be argued that appropriate formal methods may help with all of the above issues, but - in my view - most of the organisations that I see aren't yet (anywhere near) ready to go this far.]*
>
> *It may - of course - be that the organisations I have closest contact with are atypical: they are, after all, a self-selecting group. However, while I'm sure that there are many organisations that have mature processes in place for the development of real-time embedded systems, I'm equally sure that this isn't the norm.*

If we assume - for the moment - that my model is correct, how do we ensure that the situation is different in 10 years time?

*If we want to "change the embedded world" through the teaching that we are providing in universities, then I think we need to start by covering what I would see as ***core*** software engineering skills for the sector that I work in (understanding system hazards, recording requirements, use of appropriate software architectures, use of coding guidelines, code reviews, testing vs. verification, etc). On top of this, we can add formal methods - but I think we need what I would see as the core skills first.*

Nick Lusty, a safety-critical systems engineer, supports Pont's observation that the requirements documentation in projects that founder is often inadequate, http://www.systemsafetylist.org/0950.htm :

Sad to say, this is a situation that I see time and time again, and it inevitably leads to problems in validating safety critical systems. Verification of the code to low level design through low level module testing providing MC/DC coverage is easy, but demonstrating that the system validly accurately meets the unformed user requirements is much harder. Using formal methods to create the requirements is one way of addressing this, as it leaves fewer corners for ambiguities to hide. Z was used extensively for iFACTS, and on a couple of other projects I have worked on have used a variety of logic tables to specify requirements where possible.

David Mentré noted, http://www.systemsafetylist.org/0963.htm :

Use of formal methods does scale to at least hundred of thousands lines of code (at least up to 158,000 lines of code).

Here are below three examples using B Method (http://www.methode-b.com/en/), using Atelier B tool (http://www.atelierb.eu/en/), with associated cost figures (to answer your above question). In those three examples coming from subway/light rail systems, both on-board and side-way software have been formalized.

In the below examples, of course, not all functional properties have been formally verified, only those relevant to safety critical functionalities. But this is the relevant part, wouldn't you agree?

** Two first examples:*

1. Subway line 14 in Paris (project end: 1998)

2. Roissy Airport Shuttle (project end: 2006)

In case 1: 86,000 lines of Ada produced; 27,800 proof obligations; 8.1% of interactive proofs; 7.1 man.month of interactive proof time.

In case 2: 158,000 lines of Ada produced; 43,610 proof obligations; 3.3% of interactive proofs; 4.6 man.month of interactive proof time.

** Another example: New York Canarsie Line CBTC (project end: ~2006).*

The software is bigger that the previous projects, for example the on-board vital software of Canarsie CBTC is bigger that ALL vital software (on-board + wayside) of Paris subway line 14 (i.e. > 86,000 lines of code).

** Software requirements formalisation: 4 persons over 7 months (28 man.month)*

** Refinement to ADA code: 3 persons over 3 months (9 man.month)*

** (Interactive) Proof: 3 persons over 3 months (9 man.month)*

** Functional test: 3 persons over 3 months (9 man.month)*

Sources:
** Formal Methods in Industry: Achievements, Problems, Future. Jean-Raymond Abrial, ICSE'06.*

** B in Large-Scale Projects: The Canarsie Line CBTC Experience. Didier Essamé and Daniel Dollé, B 2007, LNCS 4355.*

Heath Raftery comments on the situation described by Michael Pont thus:

The scenario that plays out in my world goes like this:

1. Customer C requests doodad D to solve problem P.
2. Engineer A says right, no problem, we just need to articulate the requirements and capture them in an unambiguous way. Formal methods can help, I'll show you the way.
3. Engineer B says, no problem, in fact here's a prototype I whipped up. We're almost there.

Engineer A studied embedded development at an excellent facility and has sound knowledge of formal methods.

Engineer B taught herself programming and has been writing code since before she could drive.

4. A's manager asks how D is coming along and A says fine, we're working through the requirements.
5. B's manager asks how D is coming along and B says fine, look I've got the LEDs flashing and the relays clicking.

Guess which engineer gets rewarded?

Steve Tockey gave some of his own experiences in response, http://www.systemsafetylist.org/0977.htm :

I would like to (optimistically) extend Heath Raftery's example as follows (by the way, I refuse to refer to person B as "Engineer B" because they clearly aren't one):

Possibility A) Person B's implementation of doodad D is still little more than just flashing LEDs and clicking relays when Engineer A's solution is ready to go into production.

Engineer A's production version works essentially flawlessly.

Possibility B) Person B does provide a "production version" of doodad D however that production system gives defective output on every 32nd use and crashes--requiring a complete OS reboot--on every 64th use. Engineer A's production version works essentially flawlessly.

But will the decision makers ever even notice??? ....

I have a real-world example of a variant on this theme. Details are changed......

I worked for about 8 years at a company that makes very high-tech transportation devices. I'll use cars as an analogy, but their devices are at least an order of magnitude more complex than cars.

We start off with Car Product Line 1, which the company has been building with, say, gasoline engines for almost 20 years. There's a comprehensive suite of "automated test equipment" (ATE) software--embedded in a hardware platform--for testing Product Line 1 cars as they are being manufactured. All existing test programs were traditionally-built C code that ran on HP/UX 9. Along comes the need to produce Product Line 1 cars, but with diesel engines. The "engine simulator ATE" application for gasoline engines is 25K SLOC. The most reasonable estimate is that engine sim ATE for diesel engines will also be about 25K SLOC however code reuse is not possible for reasons that can't be elaborated here. Nonetheless, at typical programmer productivity rates and the project's allocated staffing level, that's more than a year of development. The problem is that we're already in July and the first diesel engine car will be coming through the factory the following March. We only have 8 months, not 12. They simply couldn't wait until the following July (or, realistically, much later given typical software project schedule over-runs) for the diesel version of the engine sim ATE software.

The project manager, Mike (his real name), had worked with me before on some non-related projects and was aware of my involvement in model-based development so he invited me to give the team of four a presentation on the topic. The team was intrigued with the idea that we could significantly accelerate delivery because that's exactly what was needed. Everyone agreed to take the model-based development route. Estimates derived from early modeling predicted a mid-January completion date for the model-based project. We could get it done in about 6 months, well ahead of the March need date.

A mid-level manager (to remain nameless), having experienced horrible software project delays--due to necessary debugging--in the past, insisted on having the initial code written by the end of November. This was intended to allow adequate debugging time before the need date for the first diesel car. Long story short, the requirements modeling took until the middle of October. The design modeling took until early December. When that mid-level manager visited the project in early December, he almost went into orbit when he realized that the team had not yet written even one line of production code and the project was already past the point that he had mandated for "code complete". Mike almost lost his job right then and there.

There was a slight underestimate in the project, code complete (13K SLOC) and hardware integration was completed around January 21, not January 14 as predicted back in late July. We had done all of the testing that could be done without an actual car and everything worked as expected. The engine sim ATE system then sat there until mid March, waiting for *the first diesel car. When that first diesel car was ready to be tested, both it and the ATE performed flawlessly.*

A little more than a year earlier, the corporate executive management team approved the engineering & development of Car Product Line 2. The schedule from approval to Product Line 2 car #1 rolling off the assembly line was 2.5 years. The entire ATE software suite for Car Product Line 2 was included in the schedule and needed all of the 2.5 years for development. Unfortunately, that project's original software team wasted the first 1.25 years. When the executive management did a check of the Car Product Line 2 engineering & development critical path, they realized that the ATE software team was still sitting back at the starting line. The team members had authored a few interesting technical papers and played a lot of computer games but had made zero progress on actually producing ATE software. Most of that original team got reassigned to other projects and a new team was brought in. This new team noticed that Car Product Line 1's engine sim ATE was completed in about half the time that had been predicted, and that's pretty much what they had: half the time. So I and three of the four from Car Product Line 1 engine sim ATE were brought over to get Car Product Line 2 ATE going.

There was a management mandate to "reuse as much of the Car Product Line 1 code as possible". Unfortunately, code re-use was simply not an option because for some reason Car Product Line 2 had chosen C++ on HP/UX 10 for implementation. We did reuse a little code, but only 83 SLOC. Long story short, the entire ATE suite for Car Product Line 2 was delivered 3 days ahead of the need date for car #1. Keep in mind that the full ATE suite was a far bigger job, we had 30 developers and delivered 113K SLOC. 6-8 weeks after going live on the factory floor, we met with the ATE operators to see how they liked it. Simply, they were amazed at how such a complex piece of software could work so flawlessly from the very beginning. They had set up a contest to see which operator could crash ATE and nobody had been able to.

With such fully documented, high quality code the middle managers decided they didn't need nearly as many software weenies to maintain the Product Line 2 ATE code base. In their infinite wisdom they completely ignored the fact that we had built a team that took a project in seriously deep doo doo and made it successful. Rather than find another project that was in deep doo doo and turn the extra people loose on that, the excess staff got laid off (made redundant). The team's reward for doing a great job was that most of them lost their job. Sigh...

Now, wind the clock forward about 12-15 years later. I'm no longer working at the manufacturing company. By this time they were about half way into Car Product Line 3 engineering and development. Deja vu all over again as they realized that the original ATE software team had wasted the first half of the project schedule. Again, those people got re-assigned to other projects and I got a panic call from the new ATE software team. "Aren't you the guy who bailed out the Product Line 2 ATE software project?". "Yes, why?" "We desperately need your help..."

But again, code reuse was simply not an option because the Product Line 3 ATE project had already committed to C#/.net. Nonetheless, ATE software was ready well before Product Line 3 Car #1 was in a position to be tested. And when tested, both car #1 and ATE software performed flawlessly.

One very important lesson that this company never learned was that one major reason each of these projects were able to finish on time/early was because we reduced the amount of *rework to negligible levels. Software projects at that company, like traditional software projects everywhere, suffered from 50-60% rework ("debugging"). All of the model-based ATE software projects featured peer review of the models that revealed and allowed removal of the majority of the defects before a single line of code was ever written. Rework on these projects was well under 10%, probably closer to 5%.*

The other very important lesson that the company never learned was that the other major reason for finishing on time/early was because of requirements model reuse. If you laid the three requirements models side-by-side you would notice that 80-90% of the content was identical. From a "what does it mean to test a car?" perspective, each version of ATE was largely just a minor modification of the earlier version, thus saving huge amounts of requirements development time.

In the end, my point is that the data is there. Projects have been done this way and those projects have been successful. But the business has to take the blinders off and understand what was done differently and why it made a difference. They seem to be totally incapable of this. Insert another sigh here...

I should add that what was done on these projects was not strictly "formal methods" in the sense that's being hotly debated here. We didn't use Z, VDM, or any of those formal languages. We didn't use theorem provers either. We used UML (and pre-UML because of project timing) class diagrams and state charts mostly, but we had a carefully defined and enforced (via the model inspections) single interpretation of that modeling language. Much like I mentioned in the Jeannette Wing "A Specifier's Introduction to Formal Methods" paper earlier, the modelers were using a comfortable surface syntax (UML) but there was a rigorous (albeit not exactly formally-defined) semantic to those models.

I can only speculate on the scalability of formal methods based on my experience. I suspect that they will scale just fine, provided that the people doing the majority of the "methods" work can work in comfortable surface syntaxes like UML and keep the Z, VDM, Larch, etc. stuff hidden under the covers. If anyone wants to do theorem proof of some interesting property, they are free to do so. Simply take the existing UML model and translate it into the underlying formal language equivalent and run the analysis on that. Every property proven about the formal representation has to apply to the UML version because they are equivalent semantics--they are just in a different syntax.

I don't have to speculate on the scalability of the ("semi-formal"?) model-based development process. I've personally been involved on projects that had more than 250 programmers working for about 5 years on code bases up to about 5-10 million SLOC. The projects delivered on time (or early) and the users were amazed by how few defects they encountered in actual use.

An anonymous contributor, a critical-system analyst and developer with a doctorate in computer science, contributed the following to a subgroup of discussants (private communication with an author and others):

Two observations:

1) I taught an introductory course in software engineering at the beginning Master-of-Science level for about 10 years, about 10 years ago, as an adjunct. I gave the same finite-state-machine problem for my first "programming exercise" every term. I changed the names and other data to conceal that it was the same problem. At the beginning of the time-frame, about 75% of the students got the problem "right" or "nearly right". At the end of

> *the time-frame, only about 25%. The official prerequisites to enter the program were*
>
> *a) A bachelor's degree in software engineering, computer science, information systems, computer engineering or similar.*
>
> *b) At least 2 years of practical experience working with comuters in industry or academia.*
>
> *This lack-of-knowledge of very basic logic principles of finite-state-machines is, my opinion, somewhat distressing.*
>
> *2) In 2012, we hired a second person to work along with me at [my company] in "Software Quality Assurance".*
>
> *I wrote a set of 7 questions that I asked every applicant.*
>
> *One of those questions was "does the set of initials MCDC mean anything to you?" and the follow-up question for those who answered "No" (every applicant answered No) was "Have you every worked on a team where code-coverage was measured to determine how good the testing of the actual code was?" Again, everyone answered No.*
>
> *I first did code-coverage measurement in 1972 ([with] PL/S....) and have done it since then at roughly half of the places I have worked, including here at [my company]. Yet it seems to be a mostly unknown technique.*
>
> *So, I'm not surprised at the lack on knowledge and training with-respect-to formal logic.*
>
> *My undergraduate work (math & teaching) had 1 class (symbolic logic, taught by Philosophy department, not math). None of my graduate work (computer science) had anything in logic, but that was 30 years ago. About 25 years ago, I did some formal proof-of-correctness using Gypsy for a part of an OS kernel.*
>
> *Right now, where I work, my emphasis is on formalizing requirements and ensuring traceability from requirements to tests that show that actions which should happen do, and those which should not, do not.*

Steve Tockey replied (private communication with an author and others):

> *For what it's worth, my employer [a software development company] has a Developer Testing class that I and some of my co-workers teach. We definitely talk about code coverage, and we even include a section on MCDC. So while the majority of the software practitioners probably haven't ever heard of it, those who have been through our class definitely HAVE heard of it. So there's at least a couple of hundred more people out there who aren't quite so far back in the dark ages.*
>
> *And while it wasn't MCDC in this specific case, I did teach a testing class to one of my customers: an oil company. They had a third party application that they used to help them design their "distillation columns". To manufacture a distillation column costs about $50 million each. They had used this 3rd party package to design 3-4 of these columns over the previous couple of years.*
>
> *An employee of this oil company was the technical person responsible for in-house use of the 3rd party package. He was in the testing class and learned about input domain coverage, boundary value analysis, and "all-pairs" testing. He used these techniques to test that 3rd party package. He found a number of relatively minor problems. But he was also shocked to discover a fundamental flaw in the 3rd party software that would cause it to generate an incorrect column design about 10% of the time. The design produced would look correct but would fail miserably in use, costing at least $50 million to replace the distillation column and that's forgetting lost production time and all of that.*

Even rudimentary [assessment] like this clearly isn't being done in practice…

Peter Bernard Ladkin said what he took out of Steve Tockey's tale, http://www.systemsafetylist.org/0980.htm :

The bit I take away from Steve's engaging tale .... is that

1. A lot of attention was paid to the requirements and general design engineering before code was written. And that seems to have been where most (in terms of effort) of the difficulties and potential slip-ups were resolved.

2. The specification languages used were formal (class diagrams, state charts) and the semantics was unambiguous. That is what I suggested needed to be learnt in my Logic White Paper.

I would call point 2 use of a formal method. Steve suggests it's not "in the sense ..... hotly debated here." If not, how about a suggestion of a term for "use of formal description language with an unambiguous semantics" that I and others can use?

Les Chambers recounted a tale of his own about "model-based development", and drew a number of lessons from the experience, http://www.systemsafetylist.org/0979.htm :

Story: the soul of a chemical reactor

For many years engineers have used software to enhance the performance of their machines. My first experience of this was in chemical processing (1975-1985). Working in a multidisciplinary team I developed process control software for chemical processing reactors. Put simply, we made synthetic latex, plastics, chlorine, insulating foams ... by controlling reaction kinetics with software.

Our goal was not only to make high quality product but also to maximise the yield from our reactants and to make the most efficient use of resources: energy, water, labour and so on. Given that most of our reactants were either a threat to human health or potentially explosive, all this had to be achieved with safety.

The software became the brains (nay the soul) of the plant, elevating the operator to a supervisor of automated chemical production. Software also allowed us to optimise the physical design of the plant (pipes, pumps and reactor vessels). Using smart control we could actually reduce the amount of plant hardware required without compromising safety.

From a business perspective, these applications were a resounding success. With much tighter, intelligent control we were able to produce more consistent quality at higher yields. Planned outages became less frequent as smart software predicted problems and took automatic corrective action before they escalated into a plant shutdown. As for the computers themselves, they were nothing but a tool in the service of chemistry, software was a means to an end. The programmers were actually chemical engineers on a mission. They couldn't care less about the beauty of their code, they were more interested in beautiful on-spec products. For me this was more than a job, it was a meaning of life, it was the best fun I ever had in my life and, to this day, remains the most successful suite of software applications I have ever witnessed.

Building Complex Things

This software success story was largely due to the adoption of the standard engineering approach to building a complex thing:

- *Develop a clear understanding of the problem - then document it. All projects started with something we called "English language", a clear statement of the operating discipline structured such that it could be easily transformed into a design models - something similar to what is now called Requirements State Machine Language. The English language was usually high quality because it was written by plant engineers intimately familiar with plant operations and reaction kinetics. As the plant engineers who wrote the control programs were typically not expert in the application of computers to control systems, I worked in a central support group that trained them in analysis methods, control theory and programming techniques.*

- *Apply the best science to the problem. The applicable sciences were the mathematics of control theory, and basic chemistry. Control theory, in existence for some years, was augmented with sampled-data systems theory to produce computer-based control. The chemical process technology was well established and documented by technology centres responsible for maintaining corporate memory of "the way we make chemicals".*

- *Partition the problem. Chemical processing plants could be very large and complex with thousands of sensors and final control elements. However, the control problem could be simply partitioned into various unit operations (reactor, a heat exchanger, a premix tank, scrubber, distillation column). We applied the finite state machine model to each unit operation and developed mechanisms for cooperation between unit operations based on state. This approach has since become known as "model-based development" - that is, create a model of the system that will support detailed validation of proposed behaviour before you write a line of code.*

- *Simplify. Our application language was a simplified variant of Fortran. It had no looping constructs (no do-whiles, no do-untils, certainly no go-tos). It could be taught to any engineer with foundation programming skills (some operators with no programming skills became programmers).*

- *Apply standard engineering practices - no exceptions. The control requirements of all plants were analysed using the same analysis method. All process control programs were organised in the same way. They could be easily read and understood by anyone in the world who had received basic training. These techniques were successfully rolled out in three regions of the USA and several European plants. The Asia-Pacific rollout became my responsibility - it wasn't hard, there was a strong engineering culture in place before I arrived and as this was brand-new technology, no preconceived ideas to overcome.*

- *Reuse elements of past solutions where possible. Successful control strategies that had been proven elsewhere were reused. Technology centres were tasked with making sure innovations in process control were communicated and reuse was maximised. Some programming exercises morphed into comprehensive copy and paste, with the attendant cost savings. My support group also took responsibility for "remembering neat ways of doing things," documenting them and promulgating them - complex stuff like feed-forward control using process modelling or simple stuff like* *"open the downstream valve before you start the pump - you idiot!!!"*

- *Apply strict quality control. It was easy for a newbe programmer to stray from our best-known practices, so the dos and don'ts of control system design and coding were well established, documented and rigourously enforced in requirements and code reviews. We affected a [very rigorous] regime in this respect.*

- *Perform analysis and simulation. From the beginning it was possible to test our programs via primitive simulations of plant conditions using dummy inputs. After a while we began to experiment with running our control algorithms against full-blown plant simulations. The effort required to analyse, specify and develop these simulations was roughly equivalent to that ploughed into the control program itself. The outcome was plant start-ups that took a week instead of a month; a massive cost saving.*

- *Manage risk - ask what could kill us next. Every plant I worked on presented many opportunities to screw up. The ramifications varied from destruction of plant equipment, to causing sickness or death, to triggering explosions that would not only destroy the chemical processing complex, but also the surrounding neighbourhood. There was therefore a formal approach to risk management. A team was tasked with identifying dangerous states of each plant. Code was then written to sense these conditions, abort any existing control actions and return the plant to a safe operating state.*

Benefits of the Engineering Approach

There were many positive outcomes from our engineering approach, the most telling of which was, in 10 years of working in this application domain - with at least 10 projects running concurrently at any point in time, I NEVER once heard of or experienced a project failure.

I attribute this to:

- *Customer focus. The control system software NEVER failed to meet the needs of the processing plant - mainly due to the customer being embedded in the project and the high levels of expertise brought to bear on requirements definition.*

- *Process control. The software development process was simple, well defined and rigourously enforced. We had no choice, we were always part of a larger systems project with immovable deadlines.*

- *Analysis. Analysis methods were mandated and therefore made consistent across all plants. You could write any program you liked as long as it implemented the plant control system as a set of cooperating finite state engines. Further, formal analysis of reaction kinetics and the equipment under control, followed by generation of simulations, significantly reduced the resources required for plant start-ups.*

- *Early validation with model-based development. The use of the finite state machine model allowed us to validate the overall plant control strategy, long before. Code was written. This eliminated overruns due to rework. We discovered that the most complex element of control was the logic around state transitions. This logic could be clearly stated in English and validated by engineers highly experienced in plant* *operations, but with no programming skills. If you like it allowed non-programmers to look deeper into what system was about to do and have more control over its behaviour.*

- *Quantification. Effort estimates were accurate. Using the state machine as an estimating proxy we could predict how long it would take to develop control software within a week regardless of who was performing the work. This injected welcome predictability into our projects.*

- *Documentation. The statement of requirement became the plant operating manual. Safety imperatives meant it was always kept up-to-date. Prior to plant commissioning these requirements were subject to rigourous review by process technology centres.*

- *Reuse of past solutions. A managed process for reusing innovative control strategies enhanced quality and saved money.*

- *Concern for maintainability and safety. Maintainability and safety were key issues in software development. Plants were constantly optimised and had long operational lives. Explicit requirement statements easily traceable to simple design archetypes (state machines) and implemented with simple readable code allowed anyone with process knowledge and basic programming training to enhance operations technology through software without compromising safety. There was a standing joke that after about three months from start up, you had to move the plant engineers who wrote the program on because life got incredibly boring. Often these plants started up as optimised as they were ever going to be. All the plant engineer could do was "stick his fingers in the program" (read over optimise) and screw it up. Better to move him on to another problem.*

- *Systems thinking. The software was always considered as a component of a larger system (never an end in itself). The impact of software on the chemical plant as a whole was assessed and substantial benefits flowed. Introducing software into a chemical processing plant produced emergent behavior: high quality product for one, but by far the greatest benefit came from the ability to trade-off plant hardware for smarter software. For example, before computer-control it was considered unsafe to mix certain combinations of reactants in the same reactor. The problem was solved by using premix reactors to create less volatile, intermediate products. Computer control gave us tighter control of reactant ratios allowing us to eliminate premix operations and charge heretofore dangerous chemical mixes into the same reactor, at savings of hundreds of thousands of dollars.*

Peter Bernard Ladkin said what he took away from Les Chambers's tale, http://www.systemsafetylist.org/0980.htm :

The bit I take away from Les's tale .... is

1. Same as above

2. Use of cooperating FSMs as a paradigm.

[Chambers] ".... All projects started with something we called "English language", a clear statement of the operating discipline structured such that it could be easily transformed into

a design models - something similar to what is now called Requirements State Machine Language. "

Could you call it a version of Controlled English? Was it ambiguous or was it unambiguous?

[Chambers] ".......You could write any program you liked as long as it implemented the plant control system as a set of cooperating finite state engines."

A major engineering company which produces some of the most sophisticated and expensive engineering artefacts around uses either Lustre or Statecharts for most of its SW projects and builds very successful, comparatively highly reliable SW. Both Lustre and Statecharts are based on the paradigm of communicating FSMs. With an underlying formal language expressing states and transitions. (I already knew some of this but thank you, anonymous, for expanding on it!)

Les Chambers responded to the queries, http://www.systemsafetylist.org/0981.htm :

[Ladkin] "Could you call it a version of Controlled English? Was it ambiguous or was it unambiguous?"

Unambiguous. The English was the operations manual. The operators basically watched things happen for most of the time. Much like an aircraft on automatic pilot. They were supervisors of automation. Every now and then some intervention might be required. For example, the system might have discharged a batch reactor to storage tanks, but it is still stuck in the discharge state, even though it is empty. Usually when a reactor was discharged it would automatically transition to a wait state, where it waits for the conditions that need to occur for it to be charged for the next batch. Why hasn't gone to the wait state? Okay, the operator looks up the English which is in a printout on the desk. He looks at the English language description of the logic around the state transition from discharge to wait and finds that the reactor weight reading (xxx kg) needs to be below a certain value and the flow reading in the discharge line needs to be zero. Say he knows that the weigh cells on the reactor are out of calibration. It actually is empty, because the discharge pump is still running and the flow in the discharge line is zero. So he manually forces the reactor into the wait state.

The English language [description in the operations manual], therefore had to be very specific, unambiguous and always up-to-date, reflecting the code. Failure to keep it up-to-date was a safety hazard, much like sending a pilot up with the wrong aircraft operations manual.

The useful thing about state engines is that, at the surface, they present a simple metaphor that most people can easily understand. As I think Steve mentioned the challenge is to achieve this with other formal methods - complex as you like under the hood, but simple when viewed from the outside. I can't emphasise enough that these methods saved my company a fortune. The story is available here: http://www.controlglobal.com/articles/2006/029/. They have since partnered with ABB and, I assume, are implementing similar ideas on the ABB System 800xA.

I know I am preaching to the converted, but I find it difficult to understand why any educational institution would be debating whether or not to teach these methods. It's a no-

brainer. There must be many opportunities to conduct joint research with large-scale users of control systems and control systems vendors. After all, don't we as software engineers, depend on computer scientists just as a mechanical engineers depend on the laws of physics. Unfortunately, cooperation between academia and industry is patchy. A great example of success is the Carnegie Mellon software engineering institute's work on architecture. Refer: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=58932. I found this article nothing short of inspiring.

Commentary

There are a number of themes recurring in various places in the discussion.

First, the importance of engaging a development team, including managers, who understand the applicable technical methods and what they achieve if used appropriately. Contributors relate stories of engineers who don't know MCDC assessment, who cannot handle basic state-machine concepts, who do not know how to use formal description languages, or the efficacy of unambiguous semantics.

Second, the importance of using an efficacious development method, which emphasises requirements specification and analysis. Both Steve Tockey and Les Chambers gave extended examples of software development in which the scope and the intent of the software was precisely specified, in detail, *before* the software (the "source code") writing task was commenced. Both related considerable success in doing so (Tockey in direct comparison with another approach). Similarly, Martyn Thomas referred to articles by the late Peter Amey of Praxis, which detail the success of Praxis's similar approach, of specifying formally the requirements for software (using a method called REVEAL, which results in requirements specifications in the formal language Z) before the software is written; indeed, the authors understand from Amey (personal communication) that very often the requirements specification was completed under a separate, prior contract to the software-production contract. (The importance of efficacious requirements specification and analysis is addressed in another publication: Ladkin, Sieker, 2014 to appear.)

Third, the importance of using some kind of description language for specification which is clear and has an unambiguous semantics. Tockey writes of UML with enforced unique interpretation. Chambers writes of Requirements State Machines and Unambiguous English (our coinage). An anonymous contributor writes of SCADE (which uses the Lustre state-machine language as specification) and Statecharts (although cautioning that the semantics of Statecharts can be problematic in certain cases). Amey indicates REVEAL/Z.

Fourth, the ubiquitous nature of state machines, and state machine techniques.

Contrary Views

One contributor is sceptical about the value of so-called formal methods, Derek M. Jones. He considers, as do most engineers familiar with their history, that claims of the efficacy of formal methods have historically been exaggerated. Historically, formal methods have been associated with "program proving", the attempt to prove mathematically that a particular computer program produces output with certain properties, or behaves in a certain specified way.

(Indeed, one of the authors worked for a project in the 1980's in California, in which the SRI code/ theorem prover EHDM had been used to attempt to verify the operating system of NASA's pioneering SIFT digital flight control computer. The SIFT verification project ran for some seven or eight years. The independent project assessors noted that the achievements of the project did not match what had been claimed for the efficacy of the verification attempt. For an account of the project, see Donald MacKenzie, Mechanising Proof: Computing, Risk and Trust, MIT Press, 2001. The project had involved a number of very distinguished computer scientists, who had published their work in the most well-respected journals in computer science and software engineering, the Journal of the ACM, the ACM Transactions on Programming Languages and Systems, and the IEEE Transactions on Software Engineering. The main result of the SIFT project was that formal verification of executable code written in an imperative language, "proving" code, was much harder than had been thought, and lay at that point for practical software systems beyond the state of the art in computer science. This is a significant result in computer science. It has led to significantly different approaches to the use of formal validation and verification techniques in software and digital-computer hardware. However, in the usual way of things it is socially difficult to document and publish failure as a result, so this major result is poorly categorised adn characterised in the literature. It is, however, available as a NASA Technical Report, as far as the author knows. The author remains proud to have had the opportunity to contribute to the SIFT verification effort!)

It seems Jones is concerned specifically about traditional program verification as above, in which formal logic is used to attempt to demonstrate that a program does what its specification says.

We note that this is just one of the applications of so-called formal methods, and not one of the 26 industrially mature methods enumerated in Ladkin 2012 (Functional Safety of Software-Based Critical Systems, paper to accompany Keynote Talk at the Ada Connection conference, Edinburgh, June 2012: http://www.rvs.uni-bielefeld.de/publications/Papers/LadkinAdaConnection2011.pdf). It does constitute a lot of the early work in program verification from 1969 onwards, the date of publication of Floyd-Hoare logic. The Hungarian academic Gergely Buday referred Jones to the work at http://www.nicta.com.au/pub?id=5717 which narrates an attempt at large-scale logical program verification using Isabelle/HOL, about which Jones is equally sceptical: http://shape-of-code.coding-guidelines.com/2012/05/23/would-you-buy-second-hand-software-from-a-formal-methods-researcher/ .

Given that Jones is largely concerned with this one aspect alone of formal methods, one which has not yet been industrially successful over the long term, it seems to the authors that

- it is correct that industrial-scale logical program verification is an unsolved problem, as Jones implicitly suggests; and

- this one application of formal methods is an overly narrow basis on which to dismiss all uses of formal methods, in particular the use of formal description languages with unambiguous semantics, given the affidavits here of other users of this technology such as Tockey, Thomas, SCADE users, Statecharts users, and SPARK developers and users, as well as the authors themselves;

- Jones's dismissal does not appear to constitute a sustained argument. He changes his estimate of the efficacy of formal program verification during discussion from "*[inefficacious for] anything but the smallest problem*" to "*may scale to a few thousand lines of code*". While this represents a worthy response to counter-arguments, such as from David Mentré who pointed out that the methods in the B toolset have been successfully used on between 100,000 and 200,000 lines of code, it does not represent a stable conclusion with a solid argument behind it. No matter what one's view of the matter, one may well wonder if further evidence from practical projects would lead Jones to a further upwards revision of the current limit of efficacy.

Most surprising, maybe, is the phenomenon that only one contributor is predominantly sceptical about the use of formal methods, and even he restricts his arguments to the historically-exaggerated claims for formal program verification. He does not address the efficacy for software development of expressing and analysing requirements using a formal description language and an unambiguous semantics for it. He does not address the question of demonstrating desirable properties of programs other than traditional formal verification, for example the claims by developers at Praxis (now Altran) that their techniques produce programs demonstrably free of run-time errors (Amey, *op. cit.*). He also does not address the question of exactly what properties of the train control software the Atelier B verification effort, cited by Mentré, were established by that effort.

There is, of course, some self-selection involved, both in joining the mailing list and in contributing to discussion on the point of logic in the computer science curriculum, and the efficacy of formal description languages with unambiguous semantics. Such self-selection may mitigate against the strong expression of contrary views.