Formal Methods: The Very Idea[*]
Some Thoughts About Why They Work When They Work

Daniel M. Berry[a†]

[a]Computer Science Department, University of Waterloo,
200 University Ave. W., Waterloo, Ontario N2L 3G1, Canada

The paper defines formal methods (FMs) and describes economic issues involved in their application. From these considerations and the concepts implicit in "No Silver Bullet", it becomes clear that FMs are best applied during requirements engineering. A explanation of why FMs work when they work is offered and it is suggested that FMs help the most when the applier is most ignorant about the problem domain.

1. INTRODUCTION

This paper is something that I have been meaning to write for some time now. I have been giving a talk whose title is that of this paper to a variety of audiences. In each case, the discussion generated was interesting and supplied more material for the ever growing talk. When I received the invitation to attend the Monterey Workshop on Engineering Automation for Computer-Based Systems, I saw an opportunity to present the talk to an audience of almost entirely formal methods people, including some of the pioneers. The talk was much better received than I thought it might be given its controversial nature. Moreover, all speaking participants at the workshop were required to produce a paper for the proceedings. The paper that I wanted to write is the result.

Because the paper represents more my personal opinion rather than some rigorously established scientific conclusion, the paper uses first person when referring to the author.

I have benefited particularly from an electronic discussion in 1995 with Martin Feather.

1.1. Author's Background

My current area of research is Requirements Engineering (RE) [5]. The focus of this area is on how to get requirements for software-intensive computer-based systems (SWICBSs). It is now recognized, as is explained in the body of this paper, that determining the requirements for SWICBSs is the hardest part of their development, and difficulties in determining these requirements are the source of a vast majority of errors found in delivered SWICBSs. The RE area is interested in understanding *why* a method of determining requirements works when it does and *why* a method of determining requirements fails when it does.

[*]with apologies to James H. Fetzer [15]

[†]on sabbatical leave from Computer Science Department, Technion, Haifa 32000, ISRAEL, with part of work done at GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany

1.2. Outline of Paper

This paper starts of with a brief definition of FMs. It then gives some feeling for the economics of applying FMs to the development of SWICBSs. Fred Brooks's observation of "No Silver Bullet" is recalled for what it says about the difficulty of determining SWICBS requirements. The paper offers that the most useful time to apply FMs is in the requirements engineering phase of SWICBS development. Not all applications of FMs lead to high quality SWICBSs. However, when they do succeed, there appear to be two factors working for that success, the second time phenomenon and qualities of the people who push for and engage in FMs.

So as not to lose readers who believe in FMs and who see parts of this paper as arguing against their use, please consider that I believe in FMs and use them when appropriate. I used to work for a company that sells FM technology and applies FM to clients' SWICBS development problems, including for secure operating systems. Moreover, I did some fundamental work on the underlying theory of one FM a long time ago [3]. The reader will see that I am generally in favor of FMs, but there are serious problems of which we must be aware. The paper offers some unconventional ideas as to why FMs are successful when they are.

2. DEFINITIONS OF FMs

Basically, an FM is any attempt to use mathematics in the development of a SWICBS, in order to improve the quality of the resulting SWICBS. For the purpose of this paper, I am trying to include in the realm of FMs anything anyone working in FM claims is an FM. If I have neglected to include one, my apologies. Please inform me by e-mail and include a reference to literature about it. For fuller discussions, see the papers by Hall, Leveson, and Wing [21,25,29], and papers cited therein.

There are three main groups of FMs, verification, intensive mathematical study of key problem, and refutation. Each group is described; its strengths, weaknesses and costs are considered,

2.1. Verification

The first group of FMs are those that attempt to provide a basis by which the software of a SWICBS can be formally proved to be a correct realization of a specification embodying its requirements. Strictly speaking, the code is proved consistent with a formal specification. Rarely, however, is the full proof carried out. Nevertheless, the FMs in this group all have as their goal the production of at least one part of a full proof of correctness. Within this group, there are many levels of formality and completeness. Here, by "completeness" is meant that of application and not that of a mathematical theory. Some of the FMs of this group can be characterized as some collection of levels of Table 1. In this table, the notation "P $\leftrightarrow$ C" means "partial through complete", "C" means "complete", and "P" means "partial".

In each verification level, the items in parentheses are included in what is verified to be consistent by virtue of transitivity provided by lower level proofs. Only the items outside the parentheses are directly involved in that level's proof. In Level 2, "basic correctness" means verification that the requirements specification satisfies any available independent specification of the correctness criterion, e.g., security.

A typical FM in the group described by Table 1 consists of some collection of levels. Sometimes only Level 1 and Level 7 are carried out with no verification. Rather, the doing of the formal specification allows much to be learned about the SWICBS to be developed before carrying out the actual development. Much more is learned this way than when only an informal,

1	P $\leftrightarrow$ C	formal specification of requirements
2	P $\leftrightarrow$ C	verification of consistency and basic correctness of requirements specification
3	P $\leftrightarrow$ C	formal specification of design
4	P $\leftrightarrow$ C	verification of consistency of requirements and design specifications
5	P $\leftrightarrow$ C	formal specification of code
6	P $\leftrightarrow$ C	verification of consistency of (requirements), design, and code specifications
7	C	code
8	P $\leftrightarrow$ C	verification of consistency of (requirements, design, and) code specifications and the code

Table 1

natural language specification is written. Sometimes only Level 1 is carried out for the purpose of documenting the requirements of the SWICBS, on the grounds that a formal specification is the most precise.

Applying FMs of this group drives up the cost of SWICBS development as high as 2 fold over applying normal systematic methods of writing just the code if only Levels 1 and 7 are carried out, 10 fold if Levels 1 through 4 and then 7 are carried out, and even higher if more verification is carried out. These are the rules of thumb that were applied in the company mentioned above that sold FM technology.

2.2. Intensive Study of Key Problems

The second group of FMs are the intensive mathematical study of one or more difficult aspects of the SWICBS. These are an attempt to avoid the heavy costs of the verification FMs. Rather than specifying the entire SWICBS, only the difficult or problematic parts of the SWICBS are considered. For example, if the job is to build a secure operating system that guarantees that only authorized users gain access to any specific file, instead of specifying the whole operating system and proving its security, one could focus on the security-relevant portion of the system at the specification, design, or code level, or at some combination of these. While this focusing is considerably cheaper than dealing with the full system, it is fraught with a serious danger of overlooking something that *is* security relevant. One cannot be certain that the ignored portions of the system are not security relevant, that they do not impact and they are not impacted the parts designated as security relevant and therefore under study. To prove that the ignored parts are not security relevant turns the FM into a costly verification. Nevertheless, as one gets experience with a class of applications, he or she becomes more certain about what can safely be ignored.

I would classify into this group any development in which theoretical knowledge is used to make the development of a program more systematic. The most common example of this phenomenon is compiler writing, which borrows heavily on the theory of phrase-structure grammars and has spawned its own collection of theory.

The rules of thumb that I have heard are that these intensive study FMs drive the development cost up from 2 through 5 fold, depending on the complexity of the problem and depending on

1	C	study of one difficult aspect of requirements e.g., security, safety
2	C	study of one difficult aspect of design
3	C	study of one difficult aspect of code

Table 2

how many levels of the study are carried out.

2.3. Refutation

The elements of the third group of FMs take an entirely different approach, that of refutation rather than verification [27], that is, instead of trying to prove that the SWICBS meets its requirements, one tries to refute the claim that it does. The advantage is that the correctness claim can be refuted by one counter example, while a proof must consider all possible cases. There are two kinds of refutation. One kind are those based on computable properties of some specification of the SWICBS. The second kind are those based on execution of some specification of the SWICBS on test data. Note that neither of these can be verification of correctness; correctness is not a computable property, and testing can be used to show the presence of errors, never their absence [11].

Given a specification of the complete SWICBS, as in Level 1 of the verification group of FMs, two examples of computable properties that can be used to refute the claim of correctness are:

- type checking in the specification
- cross reference checking in the specification

A type error, undefined identifier, or defined, but unused identifier can be the symptom of a more serious error, which a thinking human being should be able to spot given the evidence.

Given a specification of the complete SWICBS, as in Level 1 of the verification group of FMs, if the language of the specification is executable, e.g. OBJ [19] or INATEST [23,13] one should be able to execute the specification either with actual data or symbolically. The execution with test data may show errors in the specified SWICBS. Alternatively, one might be able to build a finite state model of the SWICBS, either directly or by simplifying the model or abstracting away some of the complexity. Execution of the finite state model with test data can show errors. In addition, if the model is finite state, there are computable checks, e.g., reachability analysis, that can be used to show the presence of problems. This group of activities is called model checking.

While locating an error by refutation is not guaranteed, in practice, model checking does expose errors, just as does execution of the finished SWICBS. However, as is shown in Section 5, it is highly preferable for an error to be exposed early in the life cycle than later after deployment of the finished SWICBS.

The refutation approaches cost that of Levels 1 and 7 of the verification group plus only 5–50% for the refutation, That is, refutation drives the cost of development up to between 2.05 and 2.50 fold, and this is cheaper than with full verification.

2.4. Programming Itself as an FM

Remember that a program itself is a formal specification. A programming language is a formally defined language with precise semantics just like Z, in fact, even more so than Z, which purposely leaves some things undefined. One could not prove the consistency of specifications and code if code were not formal. Therefore, programming itself is an FM in the sense that writing a formal specification is an FM. Remember that programming is building a theory from the programming language and library of abstractions, i.e., the ground, up, just like making new mathematics.

2.5. General Limitation of FMs

An ultimate problem with any of these FMs that are based on a specification of the SWICBS under design is the accuracy of this specification. If it does not specify what is desired, that is, it is not *right*, code that is consistent with this specification does not do what is desired. The specification could be wrong for an error of commission or an error of omission. The specification could deal with a given situation in an inappropriate way, e.g., shutting down an aircraft engine that is in an inconsistent state. The specification could fail to deal with an issue entirely or could even fail to detect the presence of the issue.

2.6. Economic Realities of FMs

For most software, it is just not worth the cost to apply FMs; one can get more than acceptable quality by inspection [14] for up to 15% more and absolutely superb results by just doing the software twice at the cost of about 100% more. However, for highly safety- and security-critical systems, for which the cost of failure is death or is considered very high, FMs are necessary to achieve the required correctness and are well worth the cost.

David Notkin [9] observes about model checking that sometimes it is necessary to make simplifying assumptions in the model to get a model tractable enough to be checked. This necessity creates a dilemma. Without simplification, the specification cannot be analyzed and critical problems might be overlooked. However, simplifications might hide critical problems, especially as abstraction is used to collapse a number of states into one. In the end, it is an issue of costs. Which problems cost more, the ones overlooked by lack of analysis or the ones overlooked by simplification?

On the other hand, there is evidence that careful use of FMs during RE of a SWICBS can eliminate enough errors from ever showing up later in the development of the SWICBS, when they are very expensive to fix, that the cost of the later development is reduced enough that the total cost of an FM-assisted development is no more or even less than than that of an unassisted development [21,22]. See Section 5 for more information about the cost to fix errors as a function of the development stage in which they are found.

3. ERRORS AND REQUIREMENTS SPECIFICATIONS

One thing that has been learned over the years is that most errors are introduced into SWICBSs during the requirements discovery, specification, and documentation stages, to the tune of between 65 and 85%. The coding stage introduces only about 25% of the errors ever introduced into a SWICBS [6]. Verification of the consistency of code to specification is by far the most expensive FM. Therefore, it is not clear how useful code verification is if only 25% of the errors are introduced during coding, and these errors are probably the easiest to fix. It seems that it is more cost effective to spend just 15% more for inspections than to spend more than 10 fold to fix errors introduced during coding. Therefore, the focus of FMs must be on requirements.

4. NO SILVER BULLET (NSB)

Not so long ago, Fred Brooks observed that, with respect to software development, “There’s no silver bullet” [8] that will suddenly and miraculously make programming fundamentally easier than it has been. He classifies software difficulties into two groups, the *essence* and the *accidents*. The essence of building software is devising the conceptual construct itself. This is very hard, because that conceptual construct is of arbitrary complexity, it must conform to the given real world, it constantly changes, and it is ultimately invisible. On the other hand, most productivity gains have come from fixing accidents. The accidents are the current technology that is used to develop the software. Examples of such accidents and their solutions are

- really awkward assembly language eliminated by high-level languages,
- severe time and space constraints eliminated by the introduction of big and fast computers,
- long batch turnaround time eliminated by time-sharing operating systems and personal computers,
- tedious clerical tasks for which tools are helpful eliminated by those tools, such as make, rcs, xref, spell, grep, fmt, and
- the drudgery of programming user interfaces eliminated by tools for building graphic interfaces such as X Windows, Java, Visual Basic and other GUI libraries.

These have been significant advances, and they have made coding significantly easier and less error prone. However, again, these advances attack only the minority of errors introduced by coding and do nothing about the essence.

Unfortunately, the essence has resisted attack. We have the same sense of being overwhelmed by the immensity of the problem and the seemingly endless details to take care of, and we produce the same kind of brain-damaged SWICBSs that makes the same stupid kind of mistakes, as we did 30 years ago! The source of these errors is that we just did not understand the conceptual construct that was to be constructed. We overlooked details or have some details wrong.

5. FMs AND THE ESSENCE OF SOFTWARE

Another way to describe the essence is “requirements”, not specifications, which are just a statement of requirements, but the requirements themselves. FMs just do not help identify requirements. They do not help us crack the essence.

There is a myth going around. Some FMs evangelists claim, “If *only* you had written a formal specification of the system, you wouldn’t be having these problems. Mathematical precision in the derivation of software eliminates imprecision.” Yes, formal specifications are extremely useful in identifying inconsistencies in requirements specifications, especially if one carries out some minimal proofs of consistency and constraint or invariant preservation. Interestingly, writing a program implementing the specification also helps identify inconsistencies in the specification; programming is another FM.

Contrary to the claim of these evangelists, FMs do not find all gaps in understanding. As Gordon and Bieman observe, omissions of functions are difficult to recognize in formal specifications [20], just as they are in programs. von Neumann and Morgenstern [28] say, "There's no point to using exact methods where there's no clarity in the concepts and issues to which they are to be applied."

Indeed, Oded Sudarsky, in a private discussion over coffee, pointed out the phenomenon of *preservation of difficulty*. Specifically, difficulties caused by lack of understanding of the real world situation are not eliminated by use of FMs; instead the misunderstanding gets formalized into the specifications, and may even be harder to recognize simply because formal definitions are harder to read by the clients. Sudarsky adds that formal specification methods just shift the lack of understanding from the implementation phase to the specification phase. The air-bubble-under-wallpaper metaphor applies here; you press on the bubble in one place, and it pops up somewhere else.

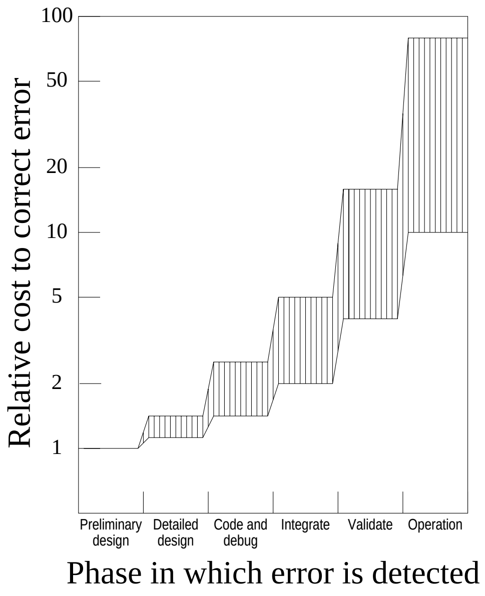

Figure 1

FMs *do* have one positive effect, and it is a big one. Use of them increases the correctness of the specifications. Therefore, you find more errors of commission at specification time than without them, saving considerable money for each bug found earlier rather than later. Remember, the cost to repair an error goes up dramatically as the project moves towards completion and beyond. Figure 1 shows a graph relating the relative cost to repair an error as a function of the SWICBS development stage in which the error is found. Note that the cost scale on the *y*-axis is logarithmic, and the graph itself looks exponential even on a logarithmic scale. It saves lots of money to find errors earlier, and FMs help find errors earlier. However, these errors are of commission rather than of omission.

In general, the more time spent studying the requirements, the easier it is to control costs. The graph in Figure 2, adapted from that by Forsberg and Mooz [16], plots data that show dramatic reductions in cost overruns of 25 NASA projects as greater portions of total project budgets are spent on the study phases. These projects include the Hubble Space Telescope, Pioneer Venus, and Voyager. Clearly, after a longer study phase,

1. the requirements specification is more complete and thus development encounters fewer surprises that would delay the project completion,
2. it is easier to more accurately estimate the total resources needed for development, or
3. both.

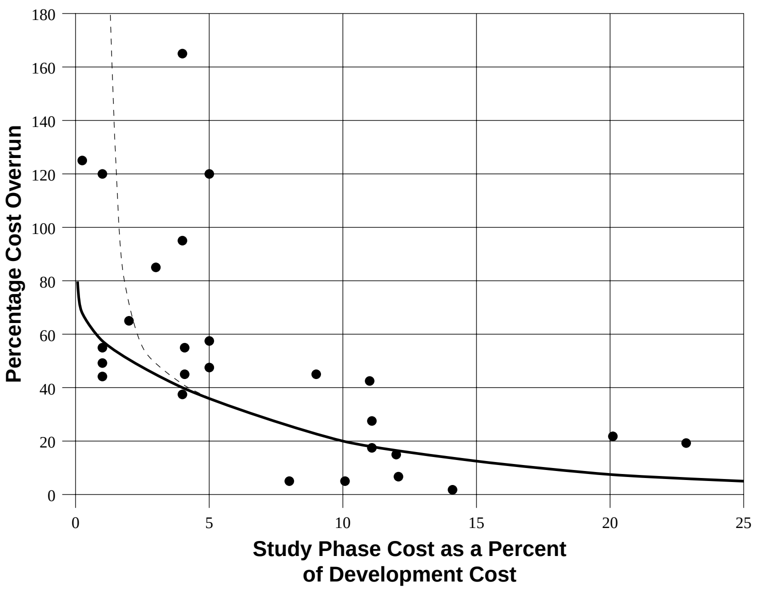

Figure 2

Another reason FMs do not help identify requirements very well is that requirements always change—it is inherent in the software—and formalization requires freezing the requirements long enough to write the specification and carry out the verifications. Meir Lehman identifies concept of E-type system [24]. It is a system that solves a problem or implements an application in some real world domain. Once installed, an E-type system becomes inextricably part of the application domain, so that it ends up altering its own requirements. As an example, consider a bank that exercises an option to automate its process and then discovers that it can handle more customers. It advertises and gets new customers, easily handled by the new system but beyond the capacity of the manual way. The bank cannot back out of automation. The requirements of the system have changed from being just optional to being required. Also, daily use of a system causes an irresistible ambition to improve it as users begin to suggest improvements. Who is not familiar with this phenomenon, either as a customer or as a developer? In fact, data show that most maintenance is not corrective, but for dealing with E-type pressures! See Figure 3. Formalization of the requirements does nothing to make the details of these kinds of changes more predictable.

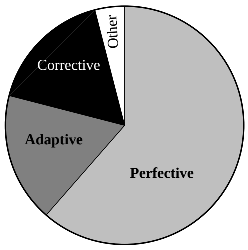

Figure 3

6. SECOND TIME PHENOMENON

In 1985, I published a paper with Jeannette Wing that suggests that FMs work, not because of any inherent property of FMs as opposed to just plain programming, which is really also an FM, but rather, because of the second time phenomenon [2]. If you do anything a second time around you do better, because you have learned from your mistakes the first time around. Indeed, Fred Brooks says: “*Plan* to throw one [the first one] away; you *will* anyway!” [7] In other words, you cannot get it right until the second time. If you write a formal specification and then you write code, you’ve done the problem formally two times. Of course, the code will be better than if you had not done the formal specification. It is the second time! Note that writing an informal specification and then writing code does not have the same effect. It is too easy to handwave and overlook details and thus fail to find the mistakes from which you learn. It has to be two formal developments, specifications or code, for the second-time phenomenon to work.

Observe how the two-time approach is requirements centered. One is not going to fix implementation errors this way, because the second time is not the same implementation as the first time. Even if they were the same, one can introduce new errors in the rewrite. The focus of the first specification or coding effort is on understanding the essence and eliminating requirements errors. The focus of the second is on implementing the understood essence. As Euripedes says, “Second thoughts are always wiser”.

7. THE IMPORTANCE OF IGNORANCE

In a recent article, “Importance of Ignorance in Requirements Engineering” [4], I report on my and Orna Berry’s experiences in practicing ignorance hiding [1] in requirements engineering. I observed that I seem to do best when I am in fact most ignorant of the problem domain. For example, I had been called in as a consultant to help a start-up write requirements for a new multi-port Ethernet switching hub. I protested that I knew nothing about networking and Ethernet beyond nearly daily use of telnet, ftp, and netfind. At one point, earlier in my life, I had worried that the ether in Ethernet cables might evaporate! Despite my ignorance, I did a superb job, in fact, better than I normally do in my areas of expertise. By being ignorant of the application area, I was able to avoid falling into the tacit assumption tarpit. The experience seems to confirm the importance of the ignorance that ignorance hiding is so good at hiding. It was clear to me that the main problems preventing the engineers at the start-up from coming together to write a requirements document were that all were using the same vocabulary in slightly different ways, none was aware of any other's tacit assumptions, and each was wallowing deep in his own pit. My lack of assumptions forced me to ferret out these assumptions and to regard the ever so slight differences in the uses of some terms as inconsistencies.

My conclusion is that every requirements engineering team requires a person who is ignorant in the application domain, the *ignoramus* of the team, who is not afraid to ask questions that show his or her ignorance, and who will ask questions about anything that is not entirely clear. It is not claimed that expertise is not needed. On the contrary, experts are needed to provide the material in which to find inconsistencies. Also, there is a difference between ignorance and stupidity; the ignoramus cannot be stupid. On the contrary, he or she must be an expert in general software system structures and how computer-based systems are built. He or she must be smart enough to catch inconsistencies in statements made by experts in fields other than his or her own, inconsistencies in their tacit assumptions, to abstract well, and to get to the bottom of things. Most importantly, he or she must be unafraid to ask so-called stupid questions to expose all tacit assumptions. (This is part of smartness since usually stupid people are afraid to ask stupid questions for fear of exposing their stupidity.)

The final recommendation is that each requirements engineering team needs at least one domain expert, usually supplied by the customer and at least one smart ignoramus.

As a consequence of these observations, resumes of future software engineers will have a section proudly listing all areas of ignorance. This is the only section of the resume that shrinks over time. The software engineer will charge fees according to the degree of ignorance: the more ignorance, the higher the fee!

Soon after publication of the Importance of Ignorance paper, I received an e-mail letter from Martin Feather. He wrote,

> I have often wondered about the success stories of applications of formal methods. Should these successes be attributed to the formal methods themselves, or rather to the intelligence and capabilities of the proponents of those methods? Typically, proponents of any not-yet-popularised approach must be skilled practitioners and evangelists to [help bring the approach] &... to our attention. Formal methods proponents seem to have the additional characteristic of being particularly adept at getting to the heart of any problem, abstracting from extraneous details, carefully organizing their whole approach to problem solving, etc. Surely, the involvement of such people would be beneficial to almost any project, whether or not they applied "formal methods."
>
> Daniel Berry's contribution to the February 1995 Controversy Corner, "The Importance of Ignorance in Requirements Engineering," provides further explanation as to why this might be so. In that column, Berry expounded upon the beneficial effects of involving a "smart ignoramus" in the process of requirements engineering. Berry argued that the "ignoramus" aspect (ignorance of the problem domain) was advantageous because it tended to lead to the elicitation of tacit assumptions. He

also recommended that “smart” comprise (at least) “information hiding, and strong typing &... attuned to spotting inconsistencies &... a good memory &... a good sense of language&...,” so as to be able to effectively conduct the requirements process.

Formal methods people are usually mathematically inclined. They have, presumably, spent a good deal of time studying mathematics. This ensures they meet both of Berry’s criteria. Mastery of a non-trivial amount of mathematics ensures their capacity and willingness to deal with abstractions, reason in a rigorous manner, etc., in other words to meet many of the characteristics of Berry’s “smartness” criterium. Further, during the time they spent studying mathematics, they were avoiding learning about non-mathematics problem domains, hence they are likely to also belong in Berry’s “ignoramus” category. Thus a background in formal methods serves as a strong filter, letting through only those who would be an asset to requirements engineering.

8. NATURE’S LAST LAUGH

Don Gause [17] points out that there are two kinds of people involved in the development of any SWICBS, developers and customers. Each person divides the universe of discourse (UoD), the domain of the SWICBS, into two parts, what he or she knows (K) and what he or she does not know (DK). The effect of these orthogonal partitions of knowledge divides the UoD into four parts as is shown in Figure 4. The problem is that we do not know the size of the DKs. We like to think that after studying the problem a long time and, the DKs have been reduced to a tiny fraction of the Ks. However, the DKs could be bigger than the Ks like the proverbial rest (as opposed to the tip) of the iceberg. Even if, in fact, the DKs are small compared to the Ks, the DKs can never be eliminated. The parts of the DKs that cannot be eliminated are called “nature’s last laugh” by Don Gause. Examples of nature’s last laugh include cold fusion and all previously accepted but now discredited theories of the universe.

		Developers	
		Know	Don’t Know
Customer & Users	Know		
	Don’t Know		Nature’s Last Laugh (DK$\times$DK)

Figure 4

The importance of the ignoramus comes through loud and clear. Every RE team requires a smart ignoramus relative to the real world domain of the system under design, who is not afraid to ask questions to reduce his or her DK in an attempt to get his or her K to include the client’s and users’ K. He or she must not be stupid; in fact, he or she must know enough about system architecture to be able to formulate enough of a model to prompt the questions. Maybe this is the role of FMs, to increase the Ks, but that is all it can be. It must be accepted that there will always be the DK, nature’s last laugh that no one will find unless someone is lucky enough to ask the right ignorant question.

9. ANOTHER EXPERIENCE

A complementary paper in this issue by David Robertson [26], considers how attempts to use FMs in the early stages of SWICBS design can fail. He describes his experiences trying to teach applied mathematicians to apply temporal logic to specifying a reactive system. The experience is more evidence of the importance of ignorance in writing specifications.

Robertson’s group was collaborating for the first time with a group of computer-using applied mathematicians that knew temporal logic theory inside and out, but had never applied this theory to specify any system. They were asked to specify in temporal logic a domain that practically invited temporal specification. As Robertson described it in e-mail to me, it “was an obvious application which could be done in a short time using simple temporal relations and forms of inference which were pedestrian by comparison to those with which the temporal logic group is familiar.” After an initial failure to specify, the mathematicians asked Robertson, with some embarrassment, to write a prototype for them. He rapidly turned out a Prolog program, of less than 100 lines of the kind that a bright second year undergraduate should be able to write. This prototype proved to be enough of a trigger, and the mathematicians are now happily turning out specifications of more complex systems.

The mathematicians simply could not take the first step without something concrete to help them. Robertson believes that the difficulty was that they lacked training in problem representation. As happens with students who are unable to apply the theory they learn to problems, the mathematicians had not developed any intuition about how to abstract away the details of a complicated problem in order to get a useful specification. Robertson believes that “it is often easier to produce the sort of idealised system I described above if you are ‘just ignorant enough’ about logic not to be drawn into too complex modelling at an early stage but ‘just smart enough’ not to make logical goofs and to be able to transfer the initial prototype to more experienced hands. People in this line of work need to retain a certain amount of wide-eyed ignorance of the domain — otherwise they would be tempted to model problems too deeply, which is fun but seldom profitable.”

10. SOCIAL PROCESSES AND FMs

It is also my belief that the proper context for FMs in the development of SWICBS is in the highly social process of requirements engineering. This recalls the 1979 DeMillo, Lipton, and Perlis paper, “Social Processes and Proofs of Theorems and Programs” [10]. They observed that mathematical proofs work because of the social processes in and around them that help to ensure that only correct theorems get published; even then they are not all correct. They argued that the proofs required by FMs applied to programming are generally carried out by grunt mathematicians working alone and without the benefit of social interaction, because, unlike publishable proofs in mathematics, proofs about programs are quite simply and frankly boring. Bored grunt mathematicians make mistakes. Proofs without social processes are not trustworthy enough for the needs of systems critical enough to justify the cost of FMs.

The verification community has actually solved this problem by replacing grunt mathematicians by theorem proving programs that never get bored and, once they have matured as programs, never make mistakes. Therefore, today, no one advocating FMs would even suggest that humans would do any of the verification, except of the most interesting theorems, for example, which would arise in an intensive study approach, described in Section 2.2. This solution is not my point. My point is of the necessity of the social processes and the usefulness of those social processes to a requirements discovery effort. Using theorem proving programs could even reduce the amount of thinking that the humans on the project do and work against discovery of missing requirements.

Indeed, inspections as described by Fagan [14], amount to an attempt to inject social processes into finding faults in developed software and related documents. Inspections set up a bit of a competition between the programmmers and the inspectors. The programmers try their damndest to write code in which the inspectors will fail to find faults, and the inspectors try their damndest to find faults in the programmers' code. Thus, inspections are fun! Moreover, inspectors are people who are not part of the development team and are thus, ignoramuses with respect to that development. All descriptions of inspection of which I am aware stress the importance of the inspectors not being part of the development team [18,12]. Arndt von Staa reminds us that independent verification and validation (IV&V) teams are no more than collections of ignoramuses, with respect to the development itself, whose duty is to find flaws in the development. Their independence and thus, ignorance, is considered essential for their success.

11. HAWTHORNE EFFECT

Ric Hehner offered another possible explanation for the FMs success stories, at least those involving a FMs expert joining a real project in an experimental application of some FM. A study performed in the 1930's in Hawthorne, Illinois showed that the mere act of studying individual behavior can impact it. A participant in an experimental application of a FM may try harder, even if he or she is not the FMs expert, and succeed simply because he or she is in an experiment and is being observed. This observation suggests turning all SWICBS developments into experimental applications of FMs, by supplying to each development project a newly graduated FMs expert itching to try out his or her favorite FM on a real project. Of course, there may not be enough FMs experts to go around!

12. CONCLUSIONS

It is my belief that FMs work when they work, not so much because of formality, but rather because of

1. what is learned when applying FMs, that can be applied in the next round of development and

2. the nature of the people who willingly and enthusiastically apply FMs.

Despite the weakness of FMs at discovering new requirements, FMs work best when they are being applied to the RE stage of SWICBS development to help understand and correct the requirements.

Probably the most important implication of the observations in this paper is a better approach to convincing industry to routinely apply FMs in the development of SWICBSs. Heretofore, we have tried to sell FMs by showing that applying FMs yields significant quality improvements even though the development costs may increase and by showing that the increased FM-induced costs are in fact less than the sum of the original development costs without FMs plus the maintenance costs that are avoided by the use of FMs. Perhaps industry is so used to considering maintenance costs as costs to develop the next version, which are necessary anyway, that it does not see the cost reduction as a benefit. Perhaps we should be selling FMs as a way to entice top-notch thinkers, super abstracters, smart ignoramuses, FMs experts, to join their development projects. Good managers know that personnel quality has a far bigger impact on a development project than any technology, such as a development method [6].

ACKNOWLEDGMENTS

I thank the anonymous referees, Joanne Atlee, Don Cowan, Steve Easterbrook, Martin Feather, Sol Greenspan, Anthony Hall, Ric Hehner, Axel van Lamsweerde, Maria Viera Nelson, Dave Robertson, John Rushby, Arndt von Staa, and Meyer Tanuan for comments on earlier drafts.

REFERENCES

1. Berry, D. M. and Berry, O., “The Programmer-Client Interaction in Arriving at Program Specifications: Guidelines and Linguistic Requirements,” *Proceedings of IFIP TC2 Working Conference on System Description Methodologies*, (Ed. E. Knuth), Kecskemet, Hungary, May 1983
2. {Berry, D. M., Wing, J. M.}, “Specification and Prototyping: Some Thoughts on Why They Are Successful,” *Proceedings of TAPSOFT Conference*, pp. 117–128, Berlin, Springer, March 1985
3. Berry, D. M., “Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language,” *IEEE Transactions on Software Engineering*, **SE-13**:2, 184–201, 1987
4. Berry, D. M., “The Importance of Ignorance in Requirements Engineering,” *Journal of Systems and Software*, **28**:2, 179–184, February 1995
5. Berry, D. M. and Lawrence, B., “Requirements Engineering,” *IEEE Software*, **15**:2, March 1998
6. Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981
7. Brooks, F. P. Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, MA, 1975
8. Brooks, F. P. Jr., “No Silver Bullet,” *Computer*, **20**:4, 10–19, April 1987
9. Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J. D., “Model Checking Large Software Specifications,” *IEEE Transactions on Software Engineering*, **SE-24**:7, 498–520, July 1998
10. DeMillo, R. A., Lipton, R. J., and Perlis, A., “Social Processes and Proofs of Theorems and Programs,” *Communications of the ACM*, **22**:5, 271–280, 1979
11. Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976
12. Ebenau, R. G. and Strauss, S. H., *Software Inspection Process*, McGraw Hill, New York, 1994

13. Eckmann, S. and Kemmerer, R. A., “INATEST: an Interactive Environment for Testing Formal Specifications,” *Proceedings of the Third Workshop on Formal Verification*, Pajaro Dunes, CA, February 1985, *Software Engineering Notes*, **10**:4, August 1985
14. Fagan, M. E., “Design and Code Inspections to Reduce Errors in Program Development,” *IBM Systems Journal*, **15**:3, 182–211, 1976
15. Fetzer, J. H., “Program Verification: The Very Idea,” *Communications of the ACM*, **31**:9, September 1988
16. Forsberg, K. and Mooz, H., “System Engineering Overview,” *Software Requirements Engineering*, Second Edition, (Eds. R. H. Thayer and M. Dorfman), IEEE Computer Society, Washington, 1997
17. Gause, D., “Understanding Requirements,” Tutorial, Colorado Springs, CO, April 1998
18. Gilb, T. and Graham, D., *Software Inspection*, Addison Wesley, Wokingham, UK, 1993
19. Goguen, J. A. and Tardo, J., “An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications,” *Proceedings Conference on Specifications of Reliable Software*, Boston, 1979
20. Gordon, V. S. and Bieman, J. M., “Rapid Prototyping: Lessons Learned,” *IEEE Software*, **15**:1, 85–95, January 1995
21. Hall, A., “Seven Myths of Formal Methods,” *IEEE Software*, **7**:5, 104–103, September 1990
22. Hall, A., “Using Formal Methods to Develop an ATC Information System,” *IEEE Software*, **13**:2, March 1996
23. Kemmerer, R. A., “Testing Formal Specifications to Detect Design Errors,” *IEEE Transactions on Software Engineering*, **SE-11**:1, January 1985
24. Lehman, M. M., “Programs, Life Cycles, and Laws of Software Evolution,” *Proceedings of the IEEE*, **68**:9, 1060–1076, September 1980
25. Leveson, N. G., “Guest Editor’s Introduction: Formal Methods in Software Engineering,” *IEEE Transactions on Software Engineering*, **SE-16**:9, September 1990
26. Robertson, D., “Pitfalls of Formality in Early System Design,” *Proceedings of the Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, October 1998, and *Science of Computer Programming*, this issue, 2001
27. Rushby, J., “Calculating with Requirements,” *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pp. 144–146, IEEE Computer Society, Annapolis, MD, January 1997
28. von Neumann, J. and Morgenstern, O., *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ, 1944
29. Wing, J. M., “A Specifier’s Introduction to Formal Methods,” *IEEE Computer*, **23**:9, September 1990