



Logic versus Magic in Critical Systems

Peter Amey

Publication notes

© Springer-Verlag
Published in Lecture Notes in Computer Science 2043
D Craeynest and A Strohmeier (Eds.):
Reliable Software Technologies – Ada-Europe 2001
6th Ada-Europe International Conference, Leuven, Belgium, May 14–18,
2001

Logic Versus Magic in Critical Systems

Peter Amey

Praxis Critical Systems, 20 Manvers St., Bath, BA1 1PX, UK
peter.amey@praxis-cs.co.uk

Abstract. A prevailing trend in software engineering is the use of tools which apparently simplify the problem to be solved. Often, however, this results in complexity being concealed or "magicked away". For the most critical of systems, where a credible case for safety and integrity must be made prior to there being any service experience, we cannot tolerate concealed complexity and must be able to reason logically about the behaviour of the system. The paper draws on real-life project experience to identify some historical and current magics and their effect on high-integrity software development; this is contrasted with the cost and quality benefits that can be made from taking a more logical and disciplined approach.

1 Introduction

We live in an irrational and magical world. While more and more aspects of our daily lives are governed by technology—itself the product of 2 millennia of rational thought and science—society is increasingly seduced by the comforting delusions of the “new age”. The lack of rational thought and numeracy in today’s society is striking. A rail crash, front page news because of its very rarity, will cause thousands to abandon the trains and drive instead. More will die as a result; however, this goes unnoticed because, perversely, road accidents are so common they don’t make the news. Opinion formers in the media wear their ignorance of science and mathematics with pride but would regard themselves as profoundly uncivilized if they didn’t have a working knowledge of Shakespeare.

Paradoxically, some of this “anti-science” may stem directly from the very successes of science and its impact on our daily lives. Most of the devices we use daily, and rely on, are now too complex for anyone but a specialist to understand. This is in marked contrast to the situation only a few years ago. I am just old enough to be of the generation where an informed amateur could expect to be able to understand the workings of all the complex machines with which they came in contact: the telephone, gramophone, vacuum cleaner and car for example. Furthermore, the reasonably physically adept would expect to be able to repair most of them as well. When such a level of understanding becomes infeasible—for example, the cellular telephone—perhaps we give up and accept the device as being “magic”. As Arthur C. Clarke observed in 1962: “*Any sufficiently advanced technology is indistinguishable from magic*”.

At first sight, we might think the world of software engineering to be poles apart from the rather sad¹ image portrayed above. After all it is a modern discipline and one which is increasingly prevalent in our lives; especially the fashionable modern bits of our lives such as the communications and entertainment industries. It is also a relentlessly binary and Boolean world where the consequences of everything we do are unavoidable, predictable and exact. Yet even here we find the language of magic. Every software product (even tiny utilities like Winzip) comes with a full set of “wizards”; search the web for a simple Windows macro tool and you find “Macro Magic”; want to re-organize your disk drive? then “Partition Magic” is the answer; Unix has its “daemons” and senior hackers are “gurus”.

This is more than just a problem of style, it shapes an environment in which we expect our problems to be magically made easier for us, an environment where rational thought and hard work is supplanted by magic wands and silver bullets.

In this paper I will illustrate some of the historical and current magics of the software engineering industry; highlight the inadequacies of these approaches (especially in the context of high-integrity systems); and show how we need to re-discover logic to make real progress in future. Comparisons with the history of aeronautical engineering are used to illustrate some of the points made.

2 An Historical Magic—Order Out Of Chaos

An early example of the desire to have our problems magically removed arose after publication of Dijkstra’s seminal article: “Go To Statement Considered Harmful” [1]. Ultimately this led to “structured programming”, which was one of the earliest attempts to solve the software crisis. It also alerted the software world to the millions of lines of “spaghetti Fortran” that existed. The magicians were quick to provide an answer in the form of code restructuring tools. Into these could be poured unstructured code and out would come structured code (a kind of software entropy reduction that is probably contrary to the second law of thermodynamics). It is worth examining one of the means by which this illusion can be achieved. I will use as an example a small program (or, more accurately, subprogram) written to implement the following specification: “*Add up the first N (positive) values of the array of integers supplied*”

Before introducing the example itself it is worth noting that even this simple English specification is ambiguous. Does it mean “traverse the array until N positive values have been found and sum them” or “traverse N elements of the array and sum any positive ones (ignoring any negative ones)”? Only more (potentially ambiguous) English or a little bit of formal mathematics can make the meaning clear. For the example I am taking the latter meaning.

¹ It is interesting to note how even this simple word has been subtly redefined. To me it is “sad” that so many people have so little interest in, or understanding of, how their world works. To a much younger generation it is the exact opposite: showing such an interest or understanding would be a prime qualification for being thought “sad”!

Some Fortran-like Ada code which implements the specification might be:

```
function Sum (A : Atype; Count : Aindex)
  return Integer
is
  Accumulator, Value : Integer;
  Index              : Aindex;

begin
  Accumulator := 0;
  Index := 1;
  <<Label1>> Value := A(Index);
             if Value <= 0 then
               goto Label2;
             end if;
             Accumulator := Accumulator + Value;

  <<Label2>> if Index = Count then
             goto Label3;
             end if;
             Index := Index + 1;
             goto Label1;

  <<Label3>> return Accumulator;
end Sum;
```

This example is not very badly structured. It has no jumps into loops and only one upward go to. Still what we want is proper structured code so into the magic tool it goes which gives us:

```
function Sum (A      : Atype;
              Count  : Aindex) return Integer
is
  IC          : Counter;
  Accumulator, Value : Integer;
  Index      : Aindex;
begin
  IC := 1;
  Accumulator := 0;
  Index := 1;
  while IC < 6 loop
    case IC is
      when 1 => Value := A(Index);
               IC := 2;
      when 2 => if Value <= 0 then
                 IC := 4;
               else
                 IC := 3;
               end if;
      when 3 => Accumulator := Accumulator + Value;
               IC := 4;
      when 4 => if Index = Count then
```

```

        IC := 6;
    else
        IC := 5;
    end if;
    when 5 => Index := Index + 1;
        IC := 1;
    when 6 => null;
    end case;
end loop;
return Accumulator;
end Sum;

```

We now have a case statement inside a while loop. Both of these constructs are pillars of structured programming orthodoxy. The code is now unquestionably “structured” but the interesting question is whether it is actually any better than it was before. Well how has this miracle been achieved? Closer examination of the restructured code shows that we have introduced an extra variable, `IC`. The convoluted semantics of code which makes excessive use of the `go to` statement arises from all the meaning being locked up in the behaviour of the processor’s instruction counter and the contents of the instructions in the memory map. The variable `IC` is simply the instruction counter in disguise. All the semantics of the program are now locked up in the behaviour of `IC` and the instructions in the case statement. We have achieved the illusion of structure but nothing has really changed. Illusions are, of course, a principal tool of magicians. The risks, incidentally, were recognized by Dijkstra himself in the same paper: “*The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.*”

Why do we allow ourselves to be fooled? Because it is easier than solving the real problem. Perhaps the contract says we must deliver structured code and the alternative is understanding sufficient graph theory to do the job properly.

The lesson of this historical example is that it is extremely hard to retrofit quality and that it is impossible to retrofit quality *automatically*. If we want a well-engineered product then we must build it well to start with. This should not be a surprise: it is a lesson written throughout the history of engineering in all its disciplines.

More contemporary magics will be revealed later on.

3 The Software Process

3.1 Development as a Process of Refinement

Whatever lifecycle we choose to use, the software development process is one of *refinement*. We start with a *need*, move through *requirements* to a *specification*, a *design*, *source code* and finally *object code*. Each of these stages is accompanied by a move from the abstract to the concrete. It may also, but does not have to be, a move

from the vague to the precise (although often confused, vagueness and abstraction are not the same thing).

Needs are inevitably abstract and are usually vague as well. We might want a flight control system, a secure internet trading system or a better mousetrap; these are all aspirations rather than artefacts and do not directly lead us to a sound implementation. To a certain extent *requirements* must also be abstract for if they become too prescriptive they risk becoming specifications; however, there is no reason why they should be vague.

Specifications are interesting. Potentially they give us our first opportunity to make a really rigorous but abstract description of required system properties. Formal specifications have a good track record for allowing early reasoning and hence early error detection (see Section 8.2 for an example). There remain reservations about the scalability of such Formal Methods and about the availability of tools and training so typical specifications today are insufficiently precise to allow any kind of rigorous reasoning about projected system behaviour.

Design is a much overlooked phase of development. The more extreme advocates of Object Oriented Programming deny that there is a design phase at all. In their concept, identification of the objects modelling the real world and their accurate implementation is sufficient to ensure that a correct system will emerge. More realistically design should be seen as a key stage in the refinement of abstract requirements and specifications to concrete code. We face real design choices here: our abstract *ordered queue* can become a *linked list*, an *array* which we regularly quick sort or even a *set of temporary files* which we merge sort together. There is no right answer and the solution is not implied by the required abstract property “ordered”. For high-integrity systems a further layer of complexity may be introduced at this stage. The relatively simple concept: “is the trigger pulled?” at the specification level may require considerable extra design to achieve the necessary level of integrity. It might become the rather more complex: “does the code show the trigger is pulled via two separate calculation methods using real and complementary arithmetic and with a checksum on each answer showing that each calculation has followed the expected processing path?”. It is this area that is most likely to cause communications difficulties between software engineers and their system counterparts who cannot understand why their relatively simple control laws are proving so hard to code.

Source code, like specifications, provides another rather interesting case. It is substantially more concrete than even a formal specification since it purports to represent instructions that will be used directly by the compiler to implement our design, fulfil our specification, embody our requirements and thus meet our needs. It even looks precise and mathematical. Therein lies the trap: the apparent precision of our source code masks considerable uncertainty. Dig a little deeper beneath the mathematical veneer of $X := X + 1;$ and we find a rather less pure world of machine representations, overflows, aliasing and function side effects. There is clear evidence that attention paid to source code in the form of reviews and walkthroughs is of considerable value; however, this is despite its lack of precision and falls short of being a precise, logical reasoning process. Like dynamic testing, we may hope to detect some shortcomings by such inspections but we cannot ensure, let alone prove, their absence unless the source language is completely unambiguous and precise.

Finally we have *object code*, something which is unequivocally concrete. There is no uncertainty about its behaviour; this is determined solely by the ones and zeros of the machine code and the mask and micro-code of the processor. Unfortunately, we cannot *reason* about it, we can only *observe* its behaviour.

3.2 Some Consequences

So overall, our typical development process never results in an artefact which is susceptible to rigorous reasoning. The first really exact representation we have is object code and that arrives late in the project life and is amenable only to observation not reasoning.

For the reasons outlined above much of the software industry seems to have abandoned attempts to reason logically about software during each stage of its development. Instead we have become extremely focussed on observing the behaviour of the finished product during dynamic testing. This mindset has become so prevalent that there is a strong current trend towards effectively eliminating all lifecycle stages between requirements and object code by the use of code generators. We are encouraged to express requirements graphically and generate the code directly from the diagrams. Since all our verification activity will consist of testing the finished code the opacity of this process does not matter. Apparently we will also save time because we will get to test faster (even though our inputs to the code generator are semantically vague). The same mindset leads to another current nostrum: that the choice of programming language is not important to the quality of the finished product.

Even widely used standards such as DO-178B [2] have gone down this route with a major emphasis on one particular form of dynamic testing and little available credit for any other contributors to quality.

The obsession with getting to test quickly is, I believe, inappropriate. If we accept the software development process as being one of decreasing abstraction then as the artefacts we deal with become more concrete, contradictions and complications overlooked in earlier stages become more apparent. We are all familiar with trying to come up with a design and finding, in the course of that process, a contradiction in the specification. Even more common is to find out during coding that some overlooked deficiency of the design makes further progress impossible. It is this process that leads to late error detection and a tendency to compromise designs by “patching” when problems are found.

It is widely agreed that correction of errors is increasingly expensive the later in the development process that they are found. A development process that largely abandons attempts to reason about the earlier stages is predestined to find most errors in the later, more expensive stages. Late error detection also adds risk, especially of cost and time overruns.

These problems are, I believe, significant in most cases. They are particularly important for the development of safety-critical and other high-integrity software because of the unique properties required of such systems.

4 High-Integrity Software

High-integrity software is software where reliability is the pre-eminent requirement. Certainly it is more important than cost, efficiency, time to market and functionality. All are important; however, for systems which must work reliably, correct behaviour is the most important property of all and achieving it will dominate the development process. Historically the most common form of high-integrity software has been safety-critical software. Where loss of life, perhaps widespread, can result from software failure, the need for high integrity is self evident; however, there is an increasing trend towards the development of systems which must be regarded as high-integrity for other reasons. The economic consequences of software failure are growing all the time. At the lower end of the scale we have the costs (direct and indirect in terms of loss of consumer confidence) of product recalls of consumer products such as cars. At the upper end we have financial systems where the potential losses are incalculably high. For example the Mondex Purse [3] will provide electronic cash with (intentionally) no audit trail, so a failure of software or failure of encryption could lead to unquantifiable financial loss or the creation of arbitrary amounts of undetectable “forged” money.

Producing such systems is challenging enough but in fact the problem is rather harder than it seems at first sight. The problem is not just to produce a reliable system but to produce a system for which a credible case can be made that it *will* be reliable in advance of its deployment and *prior to any service experience* being available. The scale of this challenge cannot be over exaggerated. For avionics systems we routinely talk of failure rates as low as 10^{-9} per flying hour. Either we are being extremely disingenuous or else we really believe we can produce systems—systems reliant on software for their correct behaviour—that will not malfunction for 114000 years after deployment! That we should believe this at all is open to question; to believe we can do so by a process of informal development followed by a period of observation of the system’s behaviour during dynamic testing borders on the absurd. The Bayesian mathematics involved is unequivocal [4, 5, 6]: we cannot provide the necessary assurance by testing alone. We may be able to produce systems which turn out to be reliable enough but we cannot produce a convincing case that they *will* be reliable enough by these means alone.

So it is clear that the development of ultra high-integrity software is a qualitatively different process, with different demands and requirements, to the development of systems with lower integrity requirements. We cannot expect to use the same processes for all systems and we certainly cannot expect to achieve high reliability just by being more careful, doing a few more walkthroughs or a bit (or even a lot) more testing. Only a process optimized for the development of high-integrity software can hope to achieve it.

5 The Need to Reason

One of the characteristics of high-integrity systems is the very large percentage of the overall effort that goes into the verification processes. For critical avionics systems it is reasonable to assume that over half of the overall development budget will be used for integration testing and for verification. Sources I respect put the figure as high as 80% [7]. This heavy weighting of effort towards the back end of the development lifecycle interacts in a malign manner with the increasing cost of error correction the later errors are found. The result is that we spend most of our time in the most expensive part of the development; an economic vicious circle that is largely responsible for the perceived high cost of high-integrity software.

Only by finding ways of bringing error detection forward can we unlock this vicious circle and produce high-integrity software at a lower cost. To bring error detection forward we must start reasoning about the characteristics of the software earlier in the development process.

5.1 Barriers to Reasoning

We can only reason logically about things which have reasonably precise meanings and which don't conceal essential information from us. Unfortunately the entire typical development process described earlier seems to be purpose-designed to prevent such reasoning. We write informal specifications, often too close to being designs to be truly specifications at all. We use notations such as UML whose semantic vagueness seems almost to be a positive virtue. Finally, we code in languages with the most obscure and imprecise semantics. So we find ourselves relying on observation of actual behaviour rather than prediction of desired behaviour.

The UML diagram seems attractive precisely because it ignores the very concrete details that will emerge later as the development process takes us relentlessly towards the absolute concrete of object code. Ignoring that detail, not by true abstraction but by vagueness, makes the diagrams easier to understand and allows them to be used as a means of communication between different stakeholders in the project. Although this sounds beneficial it is only useful if the communication that takes place is useful. If the communication succeeds because the vague semantics of the diagram allows each participant to see what they want to see rather than because they have achieved a common understanding, then no real progress has been made and the contradictions and difficulties remain concealed to emerge at some more costly stage in the future. Until this time arrives, the illusion is maintained and everyone is pleased: apparent progress is being made; the specification has been discussed and agreed by everyone; and we have tools that will let us generate code from it. The reality behind the illusion is that we have a specification which is only agreed by everyone because it contains insufficient information to allow effective debate or dissent. Furthermore, the semantic vagueness of the specification prevents fully effective code generation since we cannot go from a vague description to concrete code automatically; this would be to create order out of chaos in an even cleverer way than the spaghetti

restructuring example outlined earlier. What is likely to happen in practice is that the code generator will take some pragmatic view of what a diagram means and generate code accordingly. In many cases that view will be what was needed (but still with attendant difficulty that we must show it to be so), but in others some subtle mismatch between what was intended and what is generated may arise which will lurk awaiting detection at a later, more expensive, point in the development process.

Note that the semantic vagueness of these diagrams is very different from the, superficially similar, situation with computer-aided design tools in other engineering disciplines. The structural engineer using such a design tool will have a picture of the object he is designing on his screen. That picture may show stress levels in different colours to make them easy to understand; however, behind the scenes there is some rigorous mathematics, probably finite element analysis, going on. The coloured picture is providing an abstract view of a rigorous mathematical model. Similarly, an aerodynamicist using a computational fluid mechanics tool may look at a picture of flow fields but this too is underpinned by some heavyweight mathematics providing numerical solutions to families of partial differential equations.

In stark contrast, the user of a contemporary CASE tool is not looking at an abstract view of a precise mathematical model but at a vague illustration not underpinned by anything other than the informal design decisions of the tool vendors.

When we finally arrive at some code, whether automatically generated or not, there remains the problems of showing it to be correct to the exacting standards required for high-integrity software. Here there are further barriers to reasoning. All widely-used programming languages contain ambiguities and insecurities [8]. Ambiguities arise where the language definition permits certain implementation freedoms for the compiler writer. Typical examples are expression and parameter list evaluation orders and the method of passing subprogram parameters. Insecurities arise where language rules or coding standards cannot effectively be checked. So, for example, function side effects lead to ambiguities if evaluation order is undefined but a prohibition of function side effects simply leads to a language insecurity if there is no effective way of detecting their presence. Even Ada is not immune from these difficulties despite its attempts to wriggle out of the problem through introduction of the idea of “erroneous” programs: these are programs where ambiguity allows construction of a program of uncertain meaning, insecurity means the compiler cannot warn you about it but it is still your fault because such a program is defined to be erroneous! As a tiny example of a language ambiguity consider the completely legal Ada subprogram:

```
procedure Init (X, Y : out Integer)
is
begin
    X := 1;
    Y := 2;
end Init;
```

What, however, is the meaning of the statement: `Init(A, A);`? What value does A take? The answer depends on the order of parameter association chosen by the compiler vendor and I have found different behaviours in commercial, validated compilers. The subprogram is legal but the call to it is erroneous.

These difficulties are likely to prove an irritant and waste time during the final stages of development. A larger barrier to reasoning about source code is the wilful hiding not just of detail—which is essential for abstraction—but of information vital to the reasoning process. A significant example is the location of persistent data items or program “state”. The presence of state in a piece of software greatly adds to the complexity of understanding it, reasoning about it and even testing it. It is the presence of state that makes software behave in a non-functional manner so that, for example, the outputs obtained depend not only on the inputs provided but on the history of all previous inputs. More significantly still, it is the location of state which generates and governs the flow of information through a program. Values of state variables have to be calculated and set using information from other parts of the system and current values of state have to be conveyed from their location to the places which need those values; these are information flows. Information flows lead to couplings between components. A desired characteristic of good software designs is that they result in loose coupling of highly cohesive components; indeed this is a prime claim made for object orientation. So it is clear that the location of state has a major impact on information flow and component coupling and so should be a prime design driver. Yet OOP typically regards object state as an implementation detail and UML does not have a notation to express it. For more observations on the connection between state location, information flow and coupling see [9].

Making things disappear is another popular magical illusion. So here is the trick in a software context.

5.2 Objects and Vanishing “State”

Consider a simple subprogram to exchange its parameters.

```
procedure Swap (X, Y : in out T)
is
    Temp : T;
begin
    Temp := X;
    X := Y;
    Y := Temp;
end Swap;
```

We need a temporary variable to perform the swap operation. Where should it be located? Clearly the correct place is where it has been declared here; as a local variable of the subprogram. This is the right place because it is close to the point of use and cannot cause malign information flows because it ceases to exist as soon as the swap operation is completed. The alternative, making the temporary variable a global variable is unattractive for all the opposite reasons. It creates an unnecessary flow of information outside the swap operation, reducing its cohesion, and we have to take care that the value left in the variable does not cause a flow of information, and undesired coupling, to somewhere inappropriate. That I think is clear cut, but what happens when we introduce a nicely parameterized store object?

```

package Store
is
  procedure Put (X : in T);

  function Get return T;

end Store;

```

This provides all the encapsulation expected of object orientation and nicely hides the information about what might or might not be stored in the object itself. We can now use this object in our swap operation.

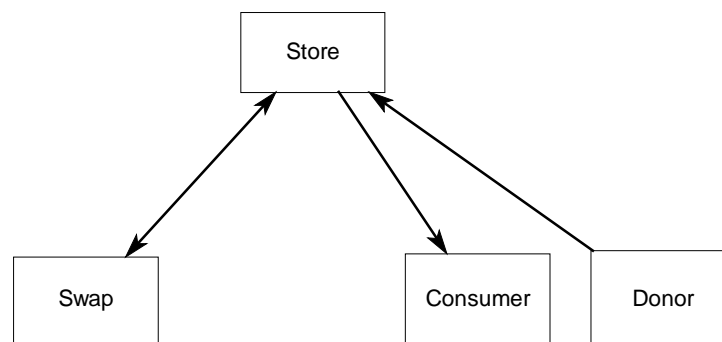
```

procedure Swap (X, Y : in out T) is
begin
  Store.Put (X);
  X := Y;
  Y := Store.Get;
end Swap;

```

The illusion is now complete, the state has vanished completely. We don't have a global variable or even a local variable any more, it has gone.

Of course like all such conjuring tricks this is just an illusion. The state has not disappeared; it is just hidden. By hiding it we make it impossible to reason about. So where is it. Well it might be inside the `Store` package or, it might be in the heap somewhere and simply be pointed at by the store object. Either way it is outside the `Swap` operation and is therefore global to it. Just as in the case of the unnecessary global variable, performing a swap operation causes unnecessary information to flow outside the operation and sets an unnecessary piece of state. The apparently atomic swap operation can now become coupled to other parts of the system. This may not matter unless that information gets used in some way. Unfortunately, another claimed benefit of object orientation—re-use—makes this highly possible. Suppose another operation called `Consumer` makes use of the store object to retrieve a value left there by `Donor`. The objects are composed thus (with the arrows showing the direction of data flows):



We can happily unit test `Store`, `Swap`, `Consumer` and `Donor`. More worryingly we may be able to integration test the `Donor-Store-Consumer` sub-system and the `Swap-Store` sub-system successfully; however, the overall behaviour of the system depends on whether a swap operation ever occurs between `Donor` setting the store value and `Consumer` reading it. If this sequence does occur then the expected value does not arrive at `Consumer`; instead, the junk value left by `Swap` is received instead—there is an unexpected coupling and information flow between `Swap` and `Consumer`. The presence of this potential coupling cannot be predicted by reasoning about the specifications of the objects involved. There is nothing about the specification of `Store` that alerts us to the potential danger. The way in which we can hide detail (to aid abstraction) but retain the ability to reason, is by having adequate abstract descriptions of the specifications of the objects we use. See for example [10].

It is the discovery of this kind of unexpected interaction during system integration that makes a major contribution to what is often called the “integration bottleneck”. Finding out why the individually thoroughly-tested objects do not always behave correctly in composition is a difficult and time-consuming task that occurs, unavoidably, late in the development with the attendant risk to schedule and budget.

So contemporary approaches to object oriented programming are potentially the biggest magic of them all. We deliberately hide all state and, worse, distribute it arbitrarily in small packets amongst the objects making up the system. We typically ignore all hierarchy between objects in the sense that some should be regarded as components of others and be encapsulated by them. We even hide the control flow through polymorphism.

Why do we do this? Because solving our problem looks so much easier without all that tiresome detail—just as designing an aeroplane would be much easier if it wasn’t for all those annoying compressible flows and that inconvenient metal fatigue.

We might very well end up with a system that happens to work by following this approach. What is certain is that we will not end up with a system for which it will be possible to construct a credible logical argument, *prior to deployment*, that the system *will* be suitable for its purpose; this, you will recall, is the especial challenge posed by high-integrity and safety-critical software. As Professor Hoare observed:

“There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.” [11]

The current state of the object oriented programming world provides an extremely powerful way of designing systems with no obvious deficiencies. Unfortunately, when operating in the rarefied atmosphere of ultra high-integrity systems it is the non-obvious deficiencies that are potentially fatal.

5.3 Other Magics

There are of course many other contemporary magics.

There is *metric magic* where we make no attempt to understand the behaviour of our system but we “measure” it instead. So we still don’t know whether it works but we know it must be better than it was before because it now measures 10.6 on some arbitrary scale where it was 4.2 before!

Then there is *process improvement magic* where we give up any attempt at understanding the system we are building and concentrate instead on improving the efficiency of the building process itself. This is like optimizing the production line rather than the product.

All of these magics have a role but as slave rather than master: control flow analysis may reveal better ways of structuring code; object orientation may provide better insights into the systems we are trying to model; metrics may indicate problem areas; and quality systems and process improvements may increase our ability repeatedly to produce correct systems (but only if we already knew how to do it once) or, more usefully, learn from both the successes and failures of others.

6 Social Issues

Many of these difficulties are compounded by non-technical and social issues. Common problem areas include:

Fashion. Choosing technologies because they are popular rather than because they are appropriate.

Low Expectations. So prevalent is buggy software that it has come to be regarded as the norm. Saying “it was only a software glitch” has almost become a socially acceptable way of excusing failure. In this environment it is hard to persuade people to make the investment necessary to do a better job.

Poor Contracting. Many conventional contract models seem ill-suited to the development of software. Time-hire terms give the developer little incentive to do a more efficient job. Fixed-price terms set the producer and customer in opposition with even mutually beneficial changes hard to negotiate. Complex software can probably only be developed effectively under terms providing partnerships and risk sharing.

Difficulty of Maintaining Standards. In a market containing more than its share of magicians and charlatans it is sometimes hard to maintain high standards of engineering integrity. If the right engineering answer is “no” then this does not mean that there will be no one willing to say “yes” (and perhaps charge you less!).

Gurus and Tactical Knowledge. Too many decisions are made at a tactical rather than strategic level. This trend is encouraged by the equating of engineering skill with experience of a particular tool or programming language. If the company software guru is an experienced C (or X or Y or Z) hacker then it is likely that he will recommend what he already knows for any job that comes along; this is the best way of preserving his guru status. Much recruitment of software engineers suffers from this kind of confusion. Agencies frequently send out lists of available engineers listing their skills as Visual Basic, C++ etc. These are not skills, they are programming languages. Listing them provides no indication about whether the engineer in question can be trusted to build a safe, software-intensive system. As an aeronautical engineer I am amused by the idea of applying for a job at Boeing or Airbus quoting my “skills” as: screwdriver, metric open-ended spanners and medium-sized hammers!

7 Can We Do Better?

We are increasingly reliant on software for our safety and for the functioning of society; the systems we seek to build are becoming increasingly complex; and testing alone cannot provide the necessary levels of assurance. It is therefore clear that we *must* seek to do better. Even if we believe we can continue to build high-integrity systems by current (or worse: by decreasingly suitable) means, the economic argument for doing a better and more rigorous job is inescapable. High-integrity software is seen to be very expensive—typically 5 times the cost of “normal” code of the same size and complexity. Much of this cost comes from the extensive testing required and the vicious circle of the high cost and risk of late error detection. We can only unlock this vicious circle by an engineering approach of “correctness by construction”.

7.1 Aerodynamics: a Lesson from History

Many other engineering disciplines have been plagued by magics of their own. Generally they have become fully established engineering disciplines when they have put magic behind them and espoused science, mathematics and logic.

For many years aeronautical engineering had the kind of split between theoreticians and practitioners that is prevalent in computer science and software engineering today. Before the Wright brothers flew at Kittyhawk on December 17th 1903 the theoretical study of aerodynamics was already very old. Aristotle, Leonardo da Vinci and Newton had all made significant observations or theoretical contributions to the science. The relationship between gas pressure and velocity was established by Bernoulli in 1738. The governing equations of frictionless low-speed incompressible flow were fully established by Euler in 1752 and then, with friction, by Navier and Stokes in 1840. However, the gulf between these researchers and the mad fools trying to make man fly was enormous. Certainly the theoreticians did not see manned flight as their goals and history clearly shows the practical aviators to be largely ignorant of the theory.

The first successful aviators, Lilienthal in Germany and the Wright brothers in the USA, leaned heavily on empirical evidence. With whirling test arms and primitive wind tunnels they developed an intuitive understanding of which aerofoil shapes were effective in practice. The rapid development in aviation that took place between the Wrights' first manned, powered flight and the end of the First World War built on this largely pragmatic approach. Designers copied each other, established a "best practice" and built largely similar aircraft. There was even an element of fashion with early monoplanes being replaced by bi- and tri-planes until the final re-emergence of the monoplane in the 1930s.

One particular aspect of this empirical approach to design is interesting: the merits of thin versus thick aerofoil sections. All early aircraft used very thin wings; this was partly intuitive because it seemed obvious that thick wings would produce more drag. The intuition was backed up by the inadequate experimental capability of the period because the effect of scale on wind tunnel results was not well understood (despite the effect being known and published in 1883 by Osborne Reynolds whose name is now used for the dimensionless number that characterizes the effect). At the low Reynolds numbers possible in early wind tunnels, thin wings were indeed superior to thick ones; a result which does not hold at the flight speeds of real aircraft of the period.

Just as this explosion of practical aviation success was taking place, advances in theoretical aerodynamics suggested that perhaps thin wings weren't so obviously correct after all. The work of Ludwig Prandtl on lifting line theory, itself built on foundations laid by Frederick Lanchester, pointed to the superiority of the thick aerofoil sections. Of all the gifted aeronautical engineers of the period only Dutchman Anthony Fokker took notice and incorporated a radically thicker wing section firstly into his Fokker Triplane and later, more significantly, into the DVII biplane. The result was an aircraft which radically outperformed the opposition and, produced in sufficient numbers, might have changed the course of the war. Significantly the terms of the 1918 armistice specified only one German aircraft type to be handed over to the allies: the Fokker DVII.

From that time onwards the aeronautical engineering industry has been characterized by the way that practical experience and theoretical, mathematical study have been made to work together. The magic of early aviation has been replaced by the logic of the current industry.

7.2 Correctness by Construction

So magic is not enough. It is attractive because it appears to make our lives simpler and offers us an easy way to achieve our goals. Unfortunately it is an illusion; the real enemy is complexity and the complexity remains even if it has been concealed by magic. Of course, mindful of Professor Hoare, we should start by trying to minimise it rather than concealing it. To use another aeronautical engineering quote (usually attributed to Bill Stout, designer of the Ford Tri-Motor), the secret of good design is to: "*simplicate, and add lightness*". The only way of dealing with the unavoidable complexity that remains is by reasoning about it.

An engineering, correctness by construction approach emphasizes the need to reason about the system and the software in it at all stages of development. The

semantic gaps between each stage in the development process are kept small so that we can show that each stage preserves the required properties from the one before. We are not aiming to meet any particular development standard but to produce a system which can be shown to be acceptable by logical reasoning; such a system can be certified to any reasonable standard. We may still have to do extensive testing, indeed standards may demand it, but we enter that test process with a system we expect to be correct, to demonstrate its correctness rather than to seek an expected crop of bugs. Structural engineers often proof-load structures but they are doing so not “to find out if it is strong enough” but to demonstrate that their stress calculations and design have been done correctly.

There is compelling evidence that this kind of reason-based, correctness by construction approach can both deliver a better product *and* reduce costs. The same study that showed the high proportion of time spent in integration and verification [7], also concluded that safety critical software cost no more than non-critical software. Their explanation is that critical systems are developed in a more rigorous manner with greater emphasis on getting things right during the early lifecycle stages. Non critical code is developed more informally and hits a bigger integration bottleneck later on. Why, as engineers, should we be surprised by the conclusion that doing things rationally and carefully is better and cheaper than doing them thoughtlessly?

8 Some Logical Successes

Two examples of the superiority of logic over magic follow. Both are SPARK² [12, 13, 14] based, not because SPARK can claim to be the world’s only logic, but because it is the logic I am most familiar with. There are many similar success stories for which space cannot be found in this paper. Please note that the examples are not intended to suggest that SPARK itself is just a superior magic. SPARK was a *necessary* part of both these successful projects but it was certainly not *sufficient* to ensure success. Neither did SPARK make complexity and difficulty magically disappear; however, it did permit sufficiently skilled engineers to reason about and deal with that complexity.

8.1 The Lockheed C130J

The Lockheed C130J or Hercules II airlifter was a major updating of one of the world’s most long-lived and successful aircraft. The work was done at the Lockheed company’s own risk. Much of the planned improvement to the aircraft was to come from the completely new avionics fit and the new software that lay at its heart. The project is particularly instructive because it has some unusual properties which provide some interesting comparisons:

- Software sub-systems developed by a variety of subcontractors using a variety of methods and languages.

² Note: the SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARCTM architecture.

- Civil certification to DO-178B.
- Military certification to UK Def-Stan 00-55 involving an extensive, retrospective IV&V activity.

For the main mission computer software Lockheed adopted a correctness by construction approach well documented in [15, 16, 17]. The approach was based on:

- Semi-formal specifications using CoRE and Parnas tables.
- “Thin-slice” prototyping of high-risk areas.
- A template-driven approach to the production of similar and repetitive code portions.
- Coding in the unambiguous language SPARK with static analysis (but not code proof) carried out prior to formal certification testing; this combination was sufficient to eliminate large numbers of errors at the coding stage; before any formal review or testing began.

This logical approach brought Lockheed significant dividends. Perhaps their most striking observation was in the reduced cost of the formal testing required for DO-178B Level A certification: “*Very few errors have been found in the software during even the most rigorous levels of FAA testing, which is being successfully conducted for less than a fifth of the normal cost in industry*”. At a later presentation [18] Lockheed were even more precise on the benefits claimed for their development approach:

- Code quality improved by a factor of 10 over industry norms for DO 178B Level A software.
- Productivity improved by a factor of 4 over previous comparable programs.
- Development costs *half* of that typical for *non safety-critical* code.
- With re-use and process maturity, a *further* productivity improvement of 4 on the C27J airlifter program.

These claims are impressive but they are justified by the results of the UK MoD’s own retrospective IV&V programme which was carried out by Aerosystems International at Yeovil in the UK. The results of this study have been widely disseminated at briefings to programme participants and the MoD. It should be remembered that the code examined by Aerosystems had already been cleared to DO-178B Level A standards which should indicate that it was suitable for safety-critical flight purposes. Key conclusions of this study were:

- Significant, potentially safety-critical, errors were found by static analysis in code developed to DO-178B Level A.
- Properties of the SPARK code (including proof of exception freedom) could readily be proved against Lockheed’s semi-formal specification; this proof was shown to be cheaper than weaker forms of semantic analysis performed on non-SPARK code.
- SPARK code was found to have only 10% of the residual errors of full Ada and Ada was found to have only 10% of the residual errors of code written in C. This is an interesting counter to those who maintain that choice of programming language does not matter and that critical code can be written correctly in any

language: the statement is probably true in principle but clearly not commonly achieved in practice.

- No statistically significant difference in residual error rate could be found between DO-178B Level A and Level B code which raises interesting questions on the efficacy of the MC/DC test coverage criterion.

As well as being an outstanding technical achievement, perhaps this project also provides an answer to another prevalent problem: the recruitment and retention of staff with suitable skills. With productivity and quality gains of this magnitude perhaps the answer is paying the right kind of engineers enough to make logic more attractive than magic.

8.2 SHOLIS

A full description of SHOLIS³ can be found in [19]. The system, with software produced by Praxis Critical Systems, was the first to meet all the principal requirements of UK Def Stan 00-55, in particular the requirements for a formal specification and code proof against that specification. The striking finding from this project was the tremendous efficiency gained by logical reasoning at an early stage in the development lifecycle. For example, the most cost-effective technique for error detection and elimination, by a considerable margin, was proof of properties of the formal, Z, specification. Two factors produced this strong gearing: the power of proof when applied to rigorous, formal descriptions; and the low cost of error correction since each corrected specification error resulted in the avoidance of wasteful re-work at some later development stage. This is the exact opposite of the vicious circle of late error detection coupled with expensive correction described earlier.

More surprisingly, code proof appeared to be substantially more effective than unit test in locating errors at the source code level. In fact unit test appears to be relatively ineffective, adding weight to the view that it is system integration that is the real challenge. The result is so convincing that Praxis has already chosen to reduce the effort allocated to unit test on subsequent SPARK projects where the relevant standard allows this flexibility. Another significant bonus from using an unambiguous programming language with suitable tool support was the feasibility of *proving* the entire SHOLIS software to be free from run-time exceptions.

The SHOLIS project shows the clear benefits of taking a disciplined, engineering approach to all lifecycle stages.

9 Conclusion

There is compelling evidence that, in its attempts to seize anything that promises to make life easier, the software industry is making its life harder. Many contemporary magics are just that: they are *not* “sufficiently advanced technology” appearing to be

³ Ship Helicopter Operating Limits Information System

magic, they have no logical or mathematical underpinnings at all. The use of such magics is foolish in general but it is potentially catastrophic to adopt such technologies for high-integrity systems. Here we cannot tolerate concealed complexity which might emerge at some unexpected and inconvenient time in the future. Having reduced it to a minimum, we must confront and reason about the complexity that remains, as other engineering disciplines have done, using mathematics and clever people. For example, the mathematics of data and information flow have been well-documented since 1985 [20] yet industry practice still regards such analysis as best being achieved by manual methods, instrumented dynamic testing [21] and code walkthroughs.

Where development processes have been created which allow software to be reasoned about *throughout all phases of development*, as in the C130J and SHOLIS examples, striking quality and economic benefits accrue. To abandon such benefits for soft, non-technical reasons (such as mistakenly equating a facility with a particular tool or programming language with engineering skill or employing the fashionable in place of the appropriate because staff “want it on their CVs”) is a self-defeating strategy that ensures that the bad will drive out the good.

In short: we must decide we want to be engineers not blacksmiths.

References:

- 1 Dijkstra, Edsger: *Go To Statement Considered Harmful*. CACM Vol 11. No. 3 March 1968, pp 147-148.
- 2 RTCA-EUROCAE: *Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B. 1992.
- 3 Ives, Blake and Earl Michael: *Mondex International: Reengineering Money*. London Business School Case Study 97/2. See <http://isds.bus.lsu.edu/cases/mondex/mondex.html>
- 4 Littlewood, Bev; and Strigini, Lorenzo: Validation of Ultrahigh Dependability for Software-Based Systems. *CACM 36(11)*: 69-80 (1993)
- 5 Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, vol. 19, no. 1, Jan. 1993, pp 3-12.
- 6 Littlewood, B: *Limits to evaluation of software dependability*. In Software Reliability and Metrics (Proceedings of Seventh Annual CSR Conference, Garmisch-Partenkirchen). N. Fenton and B. Littlewood. Eds. Elsevier, London, pp. 81-110.
- 7 Private communication arising from a productivity study at a major aerospace company.
- 8 Carré, Bernard: *Reliable Programming in Standard Languages*. In High-integrity Software. RSRE Malvern, Chris Sennett (Ed). ISBN 0-273-03158-9, 1989.
- 9 Amey, Peter: *The INFORMED Design Method for SPARK*. Praxis Critical Systems 1999.
- 10 Barnes, John: *The SPARK Way to Correctness is Via Abstraction*. ACM SIGAda 2000
- 11 Professor C.A.R. Hoare, The 1980 Turing award lecture. *The Emperor's Old Clothes*. CACM Vol. 24. No.2 February 1981. pp 75-83
- 12 Finnie, Gavin et al: *SPARK - The SPADE Ada Kernel*. Edition 3.3, 1997, Praxis Critical Systems
- 13 Finnie, Gavin et al: *SPARK 95 - The SPADE Ada 95 Kernel*. 1999, Praxis Critical Systems
- 14 Barnes, John: *High Integrity Ada - the SPARK Approach*. Addison Wesley Longman, ISBN 0-201-17517-7.
- 15 Sutton, James and Carré, Bernard: *Ada, the Cheapest Way to Build a Line of Business*". 1994

- 16 Sutton, James and Carré, Bernard: *Achieving High Integrity at Low Cost: A Constructive Approach*. 1995
- 17 Croxford, Martin and Sutton, James: *Breaking through the V&V Bottleneck*. Lecture Notes in Computer Science Volume 1031, 1996.
- 18 Sutton, James: *Cost-Effective Approaches to Satisfy Safety-critical Regulatory Requirements*. Workshop Session, SIGAda 2000.
- 19 King, Hammond, Chapman and Pryor: *Is Proof More Cost-Effective than Testing?*. IEEE Transaction on Software Engineering, Vol. 26, No. 8, August 2000, pp 675-686.
- 20 Bergeretti and Carré: *Information-flow and data-flow analysis of while-programs*. ACM Transactions on Programming Languages and Systems 1985.
- 21 Santhanam, Viswa; Wright, Peggy A.; Decker-Lindsey, Barbara: *Dataflow Coverage in the Boeing 777 Primary Flight Control Software*. Boeing 1995