ADVANCES IN SOFTWARE TECHNOLOGY SINCE 1992

AND A MODEST PROPOSAL FOR THEIR INCORPORATION INTO CERTIFICATION

John C. Knight
Department of Computer Science
University of Virginia

1. Introduction

DO-178B was published in 1992, and computing technology has changed a lot since that time. Despite the fact that the standard was written to include provision for change and many supplemental documents have been developed in the past 16 years, the advances in software technology have been so significant that it seems appropriate to investigate whether they should be accommodated in a comprehensive way and, if so, how.

In this paper, I outline a set of significant advances that have quite literally transformed the computing landscape. That transformation affects the limits of systems that can be built and how we should go about building them. The transformation is so significant that, in my opinion, standards such as DO-178B no longer serve the community in the manner intended, and any changes to it that lead to DO-178C need to accommodate the advances in technology.

The advances that I outline affect everything about a computing system. It is infeasible to treat the software as an artifact that can be developed using a standard which treats software in isolation. The approach that should be used for software development is both affected by and affects things like system hardware, system architecture, and systems-engineering practices and procedures.

I begin in section 2 with a review of major advances in software engineering. Software is affected considerably by computer system architectures, and so in section 3 I discuss advances in that area. These advances require a lot of software for their implementation, but they also affect how the software for a particular system is developed. Not surprisingly, computer system hardware impacts software and how it is built, and so I summarize advances in that field in section 4. In section 5, I summarize the impact on DO-178B and its revision. Finally, in section 6 I present an alternative way of proceedings that I refer to as DO-1743.

2. Software Engineering

2.1 Technological Changes

The greatest challenge faced by the aviation software community is the spectrum of technological advances that have come about in software engineering. DO-178C will define much about aerospace software engineering for at least a decade, and so examining the advances that have occurred since 1992 and their technical impact is vital.

In this section, I describe only some of the advances. Clearly there have been far too many to permit each to be discussed here. What I have done is present in this section a few of the major advances. They should be regarded as motivating a discussion about how the community might accommodate the entire set of advances in software engineering technology.

The *most significant change* that has occurred in software development is the nature of the predominant form of software faults that engineers have to face. In 1992, the greatest difficulty was getting the implementation to match the specification. That problem, though not solved, has been substantially mitigated by technological advances. The result is that, in general, the biggest problem faced by software engineers in the safety-critical arena is defects in requirements and specifications, not in implementations. Note carefully, that I am referring only to the *feasibility* of building implementations that match specifications closely. The available technology is not always applied and is not always successful, and as a result some development activities face an unnecessarily high rate of implementation faults.

One might reasonably ask why this change has taken place. Are software engineers somehow smarter in 2008 than they were in 1992? The change has been brought about by two major factors: (1) technological advances have provided practitioners with the better tools and techniques; and (2) the systems that the community tackles are vastly more complex. The first factor has reduced the average number of defects in implementations, and the second has increased the number in requirements and hence in specifications.

2.2 Improvements In Implementation Quality

The technological advances that have occurred since 1992 and which have helped create the possibility of implementations matching specifications closely include:

- *Practical formal specification languages*.
  In 1992 formal specification languages were in their infancy. Some had been developed, but they were little more than laboratory curiosities. To be sure some, such as VDM, had been used successfully in practical circumstances, but those activities were usually tied to a research effort. This was the case, for example, with the use of VDM to specify the PL/I semantics and the use of Z to specify the CICS business system at IBM.

  That situation has been turned around in the last 16 years. Formal languages promote accurate communication between engineers in an environment where miscommunication is often the source of difficulty. Formal specification languages also offer the possibility of automated specification analysis whereby various forms of specification defects can be identified and thereby eliminated.

  Powerful modern general specification languages include Z, RSML, Statecharts, and PVS. Despite their unfamiliarity to the average practicing engineer, they can and have been applied widely on practical projects.

  An important set of special-purpose languages has been developed for working in various domains. Those languages most relevant to the aerospace community are the languages used in model-based development. Both Simulink and SCADE are, in fact, formal specification languages although they did not arise via the normal language design route. These languages need some refinement if they are to be considered truly formal because they lack certain precision, but that is a relatively minor issue.

  Formal languages are a necessity if formal verification is to be used (see below). Thus, in terms of very high levels of assurance about implementations, formal specification languages are the starting point.

- *Software reuse*.

  In 1992, the thesis of software reuse was that it would help reduce costs. Less visible was the benefit that accrued in terms of quality by reusing something. This is, in fact, a major benefit that has helped achieve the change in the types of software fault that dominate.

  Software reuse was poorly understood in 1992. The concepts of reuse libraries, domain analysis, domain-specific languages, and application generators were known in 1992. Although these concepts were known, the extent of practical software reuse was minimal. Many efforts to build reuse libraries failed when the reusable assets were found to be either inflexible and efficient or flexible and inefficient. In addition, the goal of building generally applicable reuse libraries that served wide domains was not achieved at the time. Those reuse libraries that were successful were small and aimed at very narrow domains (simple data structures for example). Application generators were found to be very difficult to built except for extremely narrow domains, and even in those cases they produced very inefficient software.

  The state of reuse today is vastly different. Narrow domains are still the most common circumstance, but they have been made practical by narrowing system requirements down to available domains. For example, windowing systems serve narrow domains yet they serve the community well because requirements are intentionally limited to fit the available domains.

  Application generators are now commonplace. Modern application generators are also immensely more powerful and more useful than those available in 1992. Familiar examples are Simulink and SCADE. Both are in routine use in aerospace software engineering.

  Finally, in considering software reuse, the most important property to keep in mind is the quality of code developed through reuse. Application generators can be the subject of extensive analysis because they will be used by many engineers. This offers the opportunity of synthesized code being of much higher quality than human-generated code because the synthesis system can be checked and analyzed in various ways. Similarly, reuse libraries can be the subject of very high levels of checking because the library will be the subject of many uses.

- *Model-based development*.

  Model-based development is a combination of the use of a specialized (frequently domain-specific) formal specification language and an application generator. It presents a number of challenges when one tries to merge the essential characteristics of model-based development into the framework of standards such as DO-178B. Despite that, the notations used are formal and so are properly treated as such. And the creation of the code by an application generator is just that and should be considered in the context of reuse. Model-based development is not a different topic, it is a combination of a formal specification language and an application generator, and it need not be considered as a separate technology.

- *Programming languages.*

  Even in 1992, it was well known that some programming language features are error prone. Languages available at the time for use in embedded systems were many. In the early 1980s, the Department of Defense counted the number that were in use in defense systems, and stopped counting at around 500. This exercise was the motivation for the Ada project.

  Almost all of the languages in use in the DoD were ad hoc by which I mean that the language designs were not based on serious scientific principles. In part, this was because there were not a lot of scientific principles around.

With Ada ‘83, the DoD produced in one fell swoop a programming language that dealt with the vast majority of the flaws in other languages. Ada ‘83 was strongly typed, had a consistent model of real time and concurrency, could be implemented on bare machines as well as machines with operating systems, implemented separate compilation properly, had an infrastructure that allowed compilers and other tools to be checked, and had a rigorous syntax and semantics.

That is not to say that Ada ‘83 was perfect. In the years following its release and standardization, the language was found wanting, and this led to two important revisions: Ada ‘95 and Ada 2007. Many other languages have been developed since 1992 with a wide variety of goals and varying degrees of acceptance. There is, however, no question that Ada 2007 provides the best technology for developing software in a high-level language for safety-critical systems.

In an attempt to deal with the limitations of older languages, various subsets and programming guidelines have been developed. Languages such as MISRA C have tried to limit the use of C to a “safe” subset. There is some merit in doing so but far more in adopting strongly typed languages.

Measurements have been made on the fault frequency in software written in C versus software written in Ada. The evidence is anecdotal but points to the possibility of reducing defect rates by an order of magnitude just by changing languages.

- *Practical formal verification*.
  Formal verification is the creation of a proof that a software system has a specified property. Often but not always the property of interest is that the implementation matches the specification. Formal verification is also applied to hardware designs in an effort to show that a hardware design (such as the gate-level implementation of a processor) meets its specification (such as a processor’s instruction set).

  In 1992, formal verification of software was more than a laboratory curiosity. A company by the name of Computational Logic Incorporated (CLInc) applied the notion of formal verification of software to applications of interest to the DoD. They measured their costs and found that they were generally comparable to the costs of verifying software by testing.

  Since 1992, formal verification technology has made immense progress although the technology is far from being ready for routine use on all software. Theorem proving systems are much more powerful and much faster than they were in 1992. The spectrum of specifications that can be considered has been increased to include support for some aspects of real-time, floating-point, and concurrent software.

- *Model checking*.
  Model checking had only just been invented in 1992. The first paper on the subject by Ed Clarke and Alan Emmerson was published in 1989. Since 1992, the growth in both the technology and its application has been explosive.

  The principle behind model checking is to create a significant fraction of the state space of a digital system and then look for states that violate a specified property. Since the approach is based on state exploration, it is possible to derive the sequence of earlier states that lead to a final state violating the property and show the associated path in what is referred to as a counterexample.

  Model checking has been applied widely to hardware because it is easier to model large state spaces for hardware using the inherent symmetry. It has also been applied very successfully to software. One of the major applications in software is to concurrent systems to detect

deadlock and other problematic states. The application of model checking to software is essentially routine at this point.

- *Static analysis*.
  Static analysis is the examination of a formal artifact (often some software) by a machine with the goal of establishing one or more properties of the artifact. In 1992, static analysis was an established technique but it was not easily available. Many systems of the time would shower the analyst with diagnostics about the artifact essentially saying that the desired property did not hold. Frequently, the truth of the matter was that the diagnostics were false alarms.

  Since 1992, static analysis techniques have improved dramatically. The analysis tools have become more powerful, faster, easier to use, and much more accurate in the conclusions they draw.

  There has also been a move to make static analysis more effective by allowing the programmer to include specifications of properties in the program. Two examples of this type of static analyzers are SPLINT for C and the SPARK Ada toolset for a subset of Ada. This approach has created a capability in which the software can be shown to possess complex properties by analysis. By contrast with testing which shows properties only for selected values, this means that the properties hold for complete sets of states.

- *Inspections and reviews*.
  In 1992, inspections and reviews were known but not widely practiced. Michael Fagan at IBM had pioneered the use of inspections in the 1980s, but the community had mostly rejected the idea as being ineffective. Despite significant evidence from Fagan's work at IBM ($7 return for each $1 invested), almost all practicing engineers rejected the concept. The reasons had been studied earlier in the classic textbook, *The Psychology of Computer Programming* by Gerald Weinberg. Weinberg argued that the possessive attitude of software engineers about their software ensured that thorough inspections would be ineffective in practice although the opportunity for success was significant.

  Since 1992, a large number of experimental assessments of inspections have been conducted, new types of inspection have been created, and new tools developed. The result is a dramatic change in the cultural attitude in software engineering so that inspections are now viewed as an important additional verification technique. Very rigorous forms of inspection have been developed that permit high levels of assurance that software and other artifacts have requisite properties.

- *Software assessment*.
  Whether software is fit for use is perhaps the most challenging question that software engineers have to face. In 1992, the most common way that this question was answered was by testing. Software was considered fit to be used if some testing metric had been met. A common metric that was used at the time was the rate of defect detection. When new defects were detected at a rate below some threshold, the software was deemed ready. This is manifested in DO-178B by the use of the MCDC test coverage metric.

  In 1992 this approach was known to be imperfect, and around that time a lot of work was done on statistical methods of assessment. Life testing was tried but shown to be infeasible for safety-critical systems. Reliability growth models were developed but found to be generally ineffective because the models did not reflect the complexity of real systems and because the basic nature of software behavior is not considered to be stochastic.

Since 1992, the view of the community has changed considerably to a broad view of assessment. Many types of fault can be avoided by process, and other can be eliminated by analysis and informal proof. A major proponent of this type of approach was Harlan Mills in his work on the *Cleanroom* process. Mills' goal was to exclude defects by careful attention to the process, and then to assess the resulting product by life testing using an analogy with traditional manufacturing. This was suitable for the large information systems being built by IBM (Mills' employer) at the time. More recently, the ideas have been developed further by Peter Amey and his colleagues in the process they call *Correctness By Construction*. By definition, these approaches do not rely upon statistical methods to control a feedback mechanism for development. Rather they rely upon the inherent ability of careful development to produce software with known properties. Testing is not rejected, it is used as one form of evidence.

- *Development processes*.
  In 1992 the development of software processes, their management, and their role in software dependability was poorly understood. Since then, many advances have taken place in the process area, but two stand out as having the greatest impact.

  The first major advance was the appearance of a paper by Lee Osterweil of the University of Massachusetts at Amherst in 1987 entitled: "Software processes are software too." In that one paper, Osterweil showed that the description, development, management, and assessment of processes could be treated as a problem very much akin to programming. This placed process issues on a formal basis that has allowed process descriptions to become completely formal.

  The second major advance was the introduction of the Capability Maturity Model (CMM) around 1990. This was followed later by the Personal Software Process (PSP). Both represent extremely important advances in the field of software processes.

  The CMM has become a required metric in many software development and acquisition organizations. Five is the highest CMM ranking, and even now it is not held by very many organizations. Some contract offices with the DoD use a minimum CMM level as a cut-off point when selecting qualified contractors.

  There has been a lot of controversy over the CMM. Many people think that its intent is to somehow improve software quality. It is not. The goal of the CMM is to improve predictability. A CMM level 5 organization should be able to predict things like total project cost, project schedule, milestones, etc. with far greater accuracy and repeatability than a CMM level 1 (the lowest) organization.

  Not surprisingly, higher CMM levels do tend to correlate with higher software quality. Although not the intent, the higher levels of the CMM require tight control, and that leads to much better software.

  The PSP is an assessment mechanism for software engineers. It complements the CMM by allowing individual engineers to develop and measure their own engineering process in considerable detail.

- *Commercial-off-the-shelf components (COTS)*.
  COTS components are an attractive source of software because they can be purchased for a fixed cost. COTS is, of course, just a special case of reuse with (usually) large components of (usually) unknown quality.

  In 1992, COTS components were limited to commodity items produced for desk-top and similar applications. There was virtually no effort by COTS manufacturers to provide very

high-quality components. There seemed to be no market for them. The use of COTS at that time was sought after but discouraged.

Since 1992, the availability of high-quality COTS components has reached the point of eliminating the need to build significant parts of safety-critical systems. Companies such as Green Hills Software, Quantum3D and Wind River produce products that are tailored to the needs of standards such as DO-178B. COTS, thus forms a very appropriate source of reusable components even for such specialized markets as aviation. This is a major advance.

2.3 Attacking The Requirements Problem

All of the techniques discussed so far play a role in the reduction of faults in implementations. Taken together, they present a very powerful approach to dealing with the development of implementations.

As noted, however, the source of software defects in safety-critical systems that now dominates is mistakes in requirements and specifications. The problem is known to be very broad and very deep. Many of the difficulties with requirements and specifications arise because the details that are needed about a desired system are not known. Development of many aircraft computing functions begins before the associated system has been defined completely. It is known that an aircraft has to have a certain function, but the details will not be determined until parallel developments of various system elements are complete. This is an “inconvenient truth” but it is not a reason to permit the introduction of faults early in the lifecycle. Technology can deal with many of the issues.

In the period since 1992, major advances have been made in dealing with requirements and specification including the following:

- *Formal specification languages*.
  The benefits of formal specification languages outlined above are important in dealing with requirements and specification faults. One of the important analyses that is made possible by the use of formal specifications is the identification of omissions. Put simply, a formal specification can be examined to determine whether functionality has been defined for all possible inputs. This is very hard to do well with a natural language specification.

  Again, I note that progress on practical formal specification languages since 1992 has been considerable.

- *Rapid prototyping*.
  The lack of detail that hampers requirements and specification accuracy in practice can be mitigated somewhat by prototyping. Development of a prototype can be used to answer a wide range of questions such as important aspects of functionality, determination of performance adequacy, and whether systems are acceptable to users.

  This issue was recognized in 1992, and a variety of techniques for rapid prototyping were available then. Progress since 1992, however, has been substantial with new tools and techniques as well as the development of basic technology. There are several conferences each year devoted to the topic of rapid prototyping.

- *Executable specifications*.
  One approach to development of accurate specifications is to use an executable specification language. Such languages are not common, but they do exist. Simulink (and in fact any similar notation such as the SCADE Suite) is an executable specification language. It permits con-

trols engineers to develop a specification (a Simulink diagram) and then to "execute" it using what is referred to as simulation. No matter what term is used, the specification is executed to permit the developer to ascertain whether the specification actually describes what he or she wants.

Such technology was around in 1992 although it was very limited in applicability. Again, there has been a lot of change since 1992. Good example of the state of the art (more general than Simulink) are: (1) the specification technology developed at NRL based on Parnas tables; and (2) Statecharts and the associated tool Statemate. Specifications for complex embedded systems can be described quite comprehensively in both of these notations and then executed. Each provides a set of graphic icons for common peripherals such as switches, gauges, and function generators. These icons are then connected to the specification to provide user input and output. The NRL system permits graphics to be included that give the appearance of a real system such as an aircraft cockpit. Using such graphics, remarkably realistic "executions" of specifications can be undertaken.

2.4 Impact on The Future

None of the techniques discussed in this section are precluded by DO-178B. Thus, it can be argued that none need have an impact on the development of standards such as DO-178C. To an extent, this is true, but it misses the point. The point is that the advances provide a whole new world for software engineers in which huge technical areas that were poorly understood in 1992 are now reasonably well understood. Unlike the situation in 1992, many hard questions have been answered, many powerful new techniques have been developed, and many easy-to-use tools are available. Not to use this technology means engineering that is less than the state of the art, in some cases far less. Promoting and perhaps requiring the state of the art in the relevant discipline is one of the crucial roles that standards play.

3. Computer System Architecture

3.1 Technological Changes

The concept of a <u>*distributed system*</u> was certainly in place in 1992, but the technology was dominated in embedded systems by MIL STD 1553 and in non-embedded systems by simple Ethernet buses. Access was limited, network-aware devices were few and far between, there was essentially no wireless access, and the benefits of distributed systems were mostly unexplored.

Since 1992, the area has developed to a point where the benefits are extensive and widely known. Modern real-time buses such as those based on the time-triggered architecture provide very strong guarantees of important performance characteristics. The flexibility and dependability offered by distributed embedded systems makes them a desirable choice in many systems.

Wide area networks have progressed similarly. Packet-switched technologies have advanced to the point where reliable long-distance data transfers are routine, voice is carried as packets, and the cost of equipment has declined dramatically.

With the advent of distributed systems came the advent of a software architecture, known as <u>*middleware*</u>, designed to help make such systems easier to build. Middleware became available in a general way with the release of CORBA 1.0 in 1991. Since the original version of CORBA, similar middlewares have evolved including Microsoft's .Net.

3.2 Impact on The Future

The basic system architecture of many onboard and ground systems involve network technology, and this involvement is growing. It seems inevitable that aircraft will become nodes in a network that forms the backbone of the air traffic management system. This will facilitate a whole range of new services and improve the quality of many existing ones. The value of data links from air to ground is already seen in the monitoring of engine performance.

Ensuring that distributed system implementations meet necessary software dependability goals is quite difficult. Issues such as reaching consensus in the presence of faults is understood, but the available technology has to be applied systematically and carefully.

Middleware operating on embedded distributed systems is fairly common now, but it will become more so in order to support the architecture of complex applications. Again, ensuring adequate dependability of middleware implementations is a complex undertaking.

Future standards (perhaps including DO-178C) must take a position on the software aspects of computer system architecture. Although it is possible to develop software for many stand-alone avionics applications using methods that are presently used, this is **not** the case with distributed systems. It is impossible to trust a consensus algorithm, for example, based on testing. It has to be done by proof, and the standard must require it.

4. Computer System Hardware

4.1 Technological Changes

The leading processor in 1992 was the Intel 80486. A good estimate of almost any modern, general-purpose processor is that it will be about a factor of 1,000 times faster in its rate of instruction issue. Modern processors also feature larger address spaces, sophisticated virtual memory structures, on-chip caches, out-of-order execution, sophisticated pipelines including multi-threading, and multiple cores. All of these improvements have been achieved with very little increase in die size.

A similar improvement has occurred in *main memory technology*. Capacities of DRAM, SRAM and all the derivatives are much greater than they were in 1992. A conservative estimate of the improvement is a factor of 100. This advance has been accompanied by faster cycle times but memory access and cycle times have not improved nearly as much as processor clock rates.

As with main memory, there has been a dramatic increase in the size and speed of *non-volatile storage* mechanisms. Multi-gigabyte compact flash memories are now available. Other than magnetic disks and tapes, there was nothing comparable to compact flash memories in 1992. The extraordinary performance of compact flash memories suggests that it is unlikely that the rotating disk will be used for much longer in safety-critical systems.

Data communication was rudimentary in 1992. Local area networks were relatively slow and had limited capability. The primary technology was MIL STD 1553. There was no Internet as we now know it. The first Web browser, Mosaic, was released in 1993. There was no real awareness of network security issues and no serious threat. Although the world's first attack, the Morris worm, occurred in November 1988, the importance of the event was not widely appreciated for several years.

A factor in all aspects of hardware that is not well recognized yet has a significant impact in the field of safety-critical computing is that hardware is far more resilient to permanent degradation faults than it was in 1992. In 1992, the dominance and frequency of hardware degradation faults led to the need for extensive hardware replication, much of which required software monitoring and management. Operating systems and applications alike had to deal with software replication, voting, redundancy management, fault containment, and fault isolation. These issues dominated software requirements, specification, architecture, and verification.

The flip side of the reduction in degradation faults since 1992 is that modern hardware is much more susceptible to *single-event upsets* and to *Byzantine faults*. Each is now a serious problem. Single-event upsets were relatively easy to address with 1992 technology, and that technology was far less susceptible to Byzantine faults. In modern systems, software has to be involved in the management of both single-event upsets and Byzantine faults thereby, once again, leading to a situation where most aspects of software are considerably affected by hardware faults.

4.2 Impact on The Future

These hardware advances have had and will continue to have dramatic impact on aerospace computing technology. The major areas of impact are:

- The total *volume of software* in aircraft systems using digital technology is greater by orders of magnitude than was possible in 1992. There is no reason to believe this increase in volume will stop. Along with increased volume of software comes increased complexity, and the rate of increase in complexity is way above linear. In all aspects of development, this increase in volume and complexity has to be taken into account because many well-known technologies either break or become infeasible at the volumes being considered and planned.
- The total *volume of data* in all aircraft applications is also greater by orders of magnitude that was possible in 1992. This increase has enabled and will enable novel and important new applications that will then increase the demand for application software.
- Basic system architectures are affected by computer system hardware designs, and this ripples through into the *software architecture*. The predominant software architecture in 1992 was a small collection of applications executing under the control of an operating system on a uniprocessor. Modern hardware creates the opportunity to change this dramatically.
- The ability of engineers to predict the *timing of software* has declined to a point where timing predictability of modern processors and memory systems is virtually impossible unless engineers are prepared to make ultra-conservative assumptions.
- The problem of computer system *security* has arisen, and it cannot be separated from the role of any other dependability attribute or from the basic process of building the software. As with any other dependability attribute, security has to be built in. If it is not built in from the outset, whatever is done will be a less-effective compromise. Thus, security considerations both affect and are affected by the overall software development process.

  There are two major aspects of security that have to be born in mind throughout the engineering process. First, during development and operation, computer systems (both hardware, software and data) have to be protected from tampering and lack of confidentiality. Second, during operation, penetration of any aspect of the system, irrespective of whether the intruder

attempts to steal or manipulate information, has to be prevented with a high degree of assurance.

The impact of computer system hardware advances are many and they are profound. Both onboard and on the ground, enormous amounts of software will be involved in air transport. This will mean an overall different structure This requires that attention must be paid to a new level of system structure.

Finally, as a minimum, future standards must pay attention explicitly to security. The techniques required are not the same as those needed for other dependability attributes, and *it is impossible to establish security properties by testing* because these properties are not stochastic. Important security properties such as accuracy of security protocols, freedom from covert channels, and so on have to be established by proof.

5. How Does All This Affect DO-178C?

The terms of reference for SC205 (section 8) state:

> "4. Modifications to DO-178B/ED-12B should:
> 1. Strive to minimize changes to the existing text (i.e., objectives, activities, software levels, and document structure).
> 2. Consider the economic impact relative to system certification without compromising system safety.
> 3. Address clear errors or inconsistencies in DO-178B/ED-12B
> 4. Fill any clear gaps in DO-178B/ED-12B
> 5. Meet a documented need to a defined assurance benefit."

This is a very appropriate and sensible set of requirements that properly guides the committee. The notion of "minimal change" does not mean no change, however. The terms of reference are quite clear in advocating that the committee proceed in a direction that deals with all the issues facing the industry. I claim that this warrants a careful examination of how best to deal with the advances that I have outlined above.

Interestingly, DO-178B states (section 4.4, second paragraph, emphasis mine):

> "The goal of error prevention methods is to avoid errors during the software development processes that might contribute to a failure condition. The basic principle is to choose requirements development and design methods, tools, and *programming languages* that limit the opportunity for introducing errors, and verification methods that ensure that errors introduced are detected. The goal of fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process."

In many systems developed to meet DO-178B, this part of the standard is explicitly not followed. For example, as I indicated in section 2, there are programming languages that are known to eliminate the possibility of certain types of fault. By not using them, the system developers seem to be in violation of this statement in DO-178B. There are similar examples in other areas of software engineering.

With that in mind, it seems to me that it would be prudent for certification to require that applicants comply with this paragraph. Compliance does not require any specific technology. What it does require is an explanation by the applicant of why technical choices were made. Where choices are not made based on this very clearly stated goal (for example basing a choice on the available skills of the engineers involved), compliance would be denied.

6. A Modest Proposal: DO-1743

The architectures, technologies and processes of the 1980s have served us well, but they cannot deal with the challenges presented by planned systems. Modern hierarchic systems are developed with correspondingly complex processes and diverse technologies. The next generation of certification mechanisms for digital systems must reflect this in the following ways:

- Many new techniques in software engineering have become available and practical since 1992. The comprehensive and effective use of appropriate modern technology that is known to reduce the rate of software defects should be required.
- In circumstances where existing practice favors the use of technology that has been shown to be inadequate, then appropriate technology should be required explicitly. For example, there are several circumstances in which no feasible amount testing in known to be adequate. Such circumstances should be spelled out in the certification mechanism, and verification by proof should be required.
- Overall system architectures are becoming more complex as a result of the dramatic advances in computer system hardware. This complexity directly affects software and how it is built. Proper treatment of the immense and growing complexity of modern systems including topics such as distributed and network architectures must be addressed.
- Certification must include the integrated treatment of security.

A single standard such as DO-178B cannot be extended to cover all that is needed in the area of software for commercial air transports. The spectrum of technologies and the spectrum of systems being developed preclude any single approach. Any single approach would have to accommodate such a wide variety of circumstances that the standard would end up being the regulatory equivalent of a Swiss Army knife. The circumstances present in 1992 were far simpler than today. A single standard was appropriate in 1992, but the time has come to replace it with a more comprehensive approach.

To accommodate the circumstances in 2008 and beyond, I propose an approach that I will refer to as **DO-1743**. DO-1743 provides the community with a **safety-case framework** and a collection of patterns and templates such that: 1) developers wishing to build to an existing prescribed standard can continue to do so; and 2) developers building novel systems or using new technologies can take advantage of the flexibility of a safety-case-based certification strategy. Each member of the collection of patterns will have been approved by the FAA, is suitable for a class of systems that require FAA approval, and can be tailored to the needs of specific systems of interest. The members of the collection all have the same general form, but each accommodates different system and technology needs. The collection can be expanded as needed.

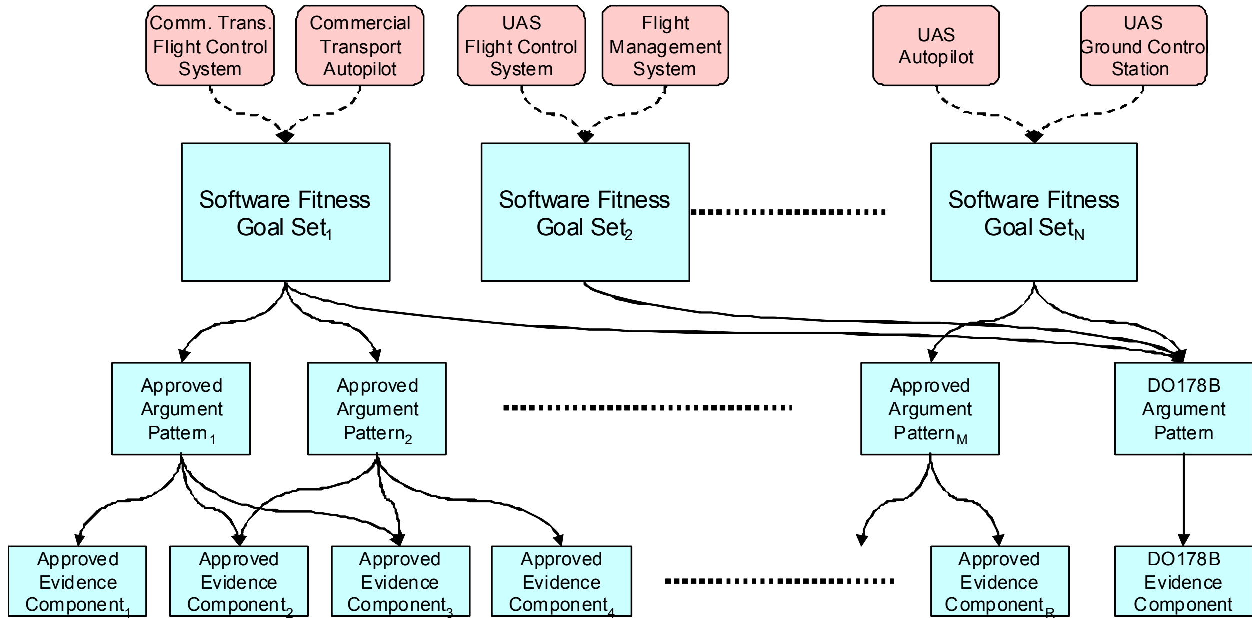

Figure 1. The DO-1743 Framework

The basic form of the DO-1743 approach is shown in Figure 1. At the top of the figure are rounded boxes labeled with typical aviation applications. They represent example systems that need to be approved using DO-1743. The dashed lines from these example systems link them to the next set of boxes in the figure, boxes with text of the form "**Software Fitness Goal Set$_i$**". Each of these is an FAA-approved set of goals for aviation software systems of particular criticality and includes the dependability requirements of that type of software. A specific goal set might require an ultra-high level of software *reliability* but make no requirement for *security*. Such a goal set would be suitable for an applicant developing a flight control system for use in a commercial air transport, and the FAA could assign this goal set to that system as part of the certification process. Similarly, a different goal set might require a high level of software *availability* together with an extreme level of *security*. Such a goal set would be suitable for an applicant developing a network-connected flight management system. The different goal sets cover all of the expected system types, and the goals are documented using the style and form of the goals used in safety cases.

For a specific system, the goal set for that system is prepared by the applicant by instantiating one of the FAA-approved goal sets using the details and parameters of the applicant's system. The choice of the goal-set pattern will be approved by the FAA for the system type involved. A flight crucial system would be required to use one goal set whereas a passenger entertainment system would be required to use another, and so on. The dashed lines between the example systems and the goal sets represent possible FAA assignments of example systems to goal sets.

In the center of Figure 1 are boxes with text of the form "**Approved Argument Pattern$_i$**" Each of these is a *pattern* or a template for a rigorous argument of the form used in safety cases. The use of the term "Approved" means that the arguments which can be built from these patterns will be acceptable to the FAA if the development of the argument has been completed successfully. An

argument for a specific system is prepared by instantiating the pattern using the details and parameters of the applicant's system. The way in which an argument is formed and stated for a security goal is vastly different from that used for availability or safety. Thus, these argument patterns are designed to provide the necessary frameworks for each of the various goal sets.

At the bottom of Figure 1 are boxes with text "**Approved Evidence Component$_i$**". Each of these is the result of some development activity that supports the argument that the goal set has been met. The results of development activities are referred to as evidence because they are used as fundamental parts of the argument. Examples of evidence components include the results of MCDC testing, the results of traceability analysis, details of a system's configuration management procedure, and so on. There are many approved evidence components.

The use of approved elements in DO-1743 does not preclude the introduction by an applicant of entirely new argument patterns or evidence. To do so, however, will require more extensive FAA review of the resulting argument. Using FAA-approved elements does not preclude the introduction of minor changes or additions in order to accommodate the specifics of particular systems. Finally, the use of FAA-approved elements provides no guarantees of approval, but it does reduce the chances of rejection and streamlines the review process because of the inherent familiarity that will arise in DERs.

Within the DO-1743 architecture there is a special case, and others can be added if that is deemed necessary. The special case is shown on the right of Figure 1, and it is the boxes with labels that include DO-178B. The special case has the same form as the rest of the figure but the details are specific to the DO-178B standard. This special case is, in fact, the safety-case pattern that corresponds to the standard. In other words, by following DO-178B, the applicant is implicitly complying with the associated safety case. This special case in DO-1743 is present to allow the continued use of DO-178B as deemed appropriate by the FAA after the transition to DO-1743. A special case could be added quite easily for DO-178C thereby permitting its use immediately it is available. The general structure of DO-1743 and its accommodation of DO-178B and DO-178C allows the completion of DO-178C to be undertaken **without the current pressure to make DO-178C completely comprehensive**.

The DO-1743 approach has the following nine important properties:

1. It can accommodate all advances in software technology that might be of value to the aviation community. This accommodation is achieved by modifying existing argument patterns or evidence components as needed. Different argument patterns can use different technologies without interfering with each other.

2. It includes DO-178B in such a way that the existing standard can be used for as long as the FAA approves its use yet compliance will be for DO-1743.

3. It makes provision for the inclusion of DO-178C once it is complete and permits its use on any systems for which it is deemed appropriate, again compliance being with DO-1743. The inclusion of DO-178C merely requires the addition of a goal set, an argument pattern, and evidence components as defined by DO-178C.

4. It provides a mechanism for the FAA to require certain combinations of technology for certain purposes as the agency sees fit to protect the flying public. This is achieved by associating

particular goal sets with particular argument patterns that are themselves associated with particular evidence components. Since a particular system will have to meet a particular goal set, the necessary technology will be implied. This provides a mechanism, for example, whereby formal methods can be required for certain kinds of systems such those using distributed agreement.

5. It provides flexibility so that, if desired, an applicant can choose technology and processes suitable for the system the applicant is building without the applicant being forced to fit into a predefined set of process and technology requirements. The argument patterns are not the only arguments that can be used. Applicants can create their own.
6. It incorporates the modern notion of safety cases as its basic structure.
7. It addresses the issues raised in the NRC Committee Report entitled “Software for Dependable Systems: Sufficient Evidence?”. The underlying technique in DO-1743 is that of safety cases and that is essentially what the NRC report suggests.
8. It can be applied to ground systems immediately and without modification. The associated goal sets can be added as needed if they do not exist along with the required argument patterns and evidence components. To the extent that it is appropriate, this will allow ground and airborne software systems to be developed differently but within the same regulatory framework.
9. It can be applied to unmanned air systems immediately and without modification.

The safety case element of DO-1743 is very similar to (arguably identical to) the U.K. MoD Defence Standard 00-56 in which applicants have a single requirement, namely to deliver a system safety case. In practice, the 00-56 approach has two difficulties that DO-1743 solves:

1. No specific process details about how to comply with the standard are provided in 00-56, and so developers are unsure of what is required to meet the standard. This problem is solved in DO-1743 by the use of goal sets created by the FAA, FAA-approved argument patterns, and FAA-approved evidence components.
2. The cost of transition from older standards such as MoD Defence Standard 00-55 and international standards such as DO-178B are large but unknown when moving to 00-56. The result is that working to the 00-56 standard is a substantial business risk. This problem is solved in DO-1743 by the inclusion of an argument pattern and evidence components that derive from DO-178B. The applicant, therefore, has a low risk starting point from which they are able to advance into the more sophisticated approaches as they see fit.