A direct path to dependable software

Citation	Jackson, Daniel. "A direct path to dependable software." Commun. ACM 52.4 (2009): 78-88.
As Published	http://dx.doi.org/10.1145/1498765.1498787
Publisher	Association for Computing Machinery
Version	Author's final manuscript
Citable link	http://hdl.handle.net/1721.1/51683
Terms of Use	Attribution-Noncommercial-Share Alike 3.0 Unported
Detailed Terms	http://creativecommons.org/licenses/by-nc-sa/3.0/

A Direct Path To Dependable Software

Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

What would it take to make software more dependable? Until now, most approaches have been *indirect*: some practices – processes, tools or techniques – are used that are believed to yield dependable software, and the argument for dependability rests on the extent to which the developers have adhered to them. This article argues instead that developers should produce *direct* evidence that the software satisfies its dependability claims. The potential advantages of this approach are greater credibility (since the argument is not contingent on the effectiveness of the practices) and reduced cost (since development resources can be focused where they have the most impact).

1 Why We Need Better Evidence

Is a system that never fails dependable? Not necessarily. A dependable system is one you can *depend* on – that is, in which you can place your reliance or trust. A rational person or organization only does this with *evidence* that the system's benefits outweigh its risks. Without such evidence, a system cannot be depended upon, in the same way that a download from an unknown website cannot be said to be 'safe' just because it happens not to harbor a virus.

Perhaps in the future we will know enough about software development practices that the very use of a particular technique will constitute evidence of the quality of the resulting software. Today, however, we are far from that goal. Although individual companies can predict defect rates within product families based on historical data, there is barely any industry-wide data collection and analysis.

Contrast software systems with cars, for example. In the US, the National Highway Traffic Safety Administration (NHTSA) maintains several databases that include records of all fatal accidents – approximately 40,000 a year – and data about how particular models fare in crashes. Researchers can use this data to correlate risk with design features; one study [55], for example, found that pickup trucks pose higher risks to drivers of other vehicles because of their chassis stiffness and height. NHTSA also receives data from auto companies regarding warranty claims, defects and customer complaints (and made it publicly available until a regulatory change in 2004 [8]). The National Transportation Safety Board, best known for its work in aviation, analyzes highway accidents too and issues reports on, amongst many other things, the efficacy of safety devices such as seatbelts and airbags.

The software industry has no comparable mechanism, and as a society we have almost no data on the causes or effects of software failure. Producers of software cannot benefit from data about the industry as a whole that would improve their designs and development strategies, and consumers cannot use such data to make informed purchasing decisions. Where data is collected, it is often suppressed; many companies, for example, withhold even basic information about the number and severity of defects when issuing patches that purport to resolve them. No government agency is charged with recording fatal accidents due to software or investigating software failures, and when an accident report does implicate software, it rarely includes enough information to allow any general lessons to be learnt.

Over the last few decades, we have developed approaches and technologies that can dramatically improve the quality of software – better platforms (safe programming languages, operating systems with address space separation, virtual machines), better infrastructure (configuration control, bug tracking, traceability), better processes (spiral and agile models, prototyping, etc), better tools (integrated environments, static analyzers, model checkers) – and we have made progress in understanding fundamentals (in problem structuring, design modeling, software architecture, verification, testing, and so on). All of these advances can be misused, however, and none of them guarantees success. The field of empirical software development is attempting to fill the gap and provide scientific measures of efficacy. But there is still no evidence that is compelling enough that simply using a given approach establishes with confidence the quality of the resulting system.

Many certification standards were devised with the good intent of enforcing best practices but have the opposite effect. Instead of encouraging the selection of the best tool for the job, and directing attention to the most critical aspects of a system and its development, they impose burdensome demands to apply the same – often outdated – techniques uniformly, and result in voluminous documentation of questionable value. Microsoft, for example, spends a fortune to obtain Common Criteria security certification for its operating systems, even though its internally-devised mitigations are known to be cheaper and more effective.

Government agencies are often in the unfortunate position of having to evaluate complex software systems solely on the basis of evidence that some process was adhered to and some amount of testing was performed. Not surprisingly, certified systems sometimes fail catastrophically; a particularly tragic example is the failure of an FDA-certified radiation therapy machine in Panama in 2001 [30], in which fatal overdoses resulted from poorly engineered software in an incident reminiscent of the Therac failures of fifteen years before [41]. Even the most highly regarded standards demand expensive practices whose value is hard to determine. DO178B, for example, the safety standard used in the US for avionics systems, requires a level of test coverage known as MCDC that is extremely costly and whose benefits studies have yet to substantiate [22].

A very different approach, sometimes called 'goal-based' or 'case-based' certification, is now gaining currency [5], in which no particular practices – processes, tools or techniques – are mandated, and there is no implicit reliance on (missing!) evidence that such practices are effective. Instead, the developer is called upon to provide *direct* evidence that the particular system being delivered satisfies its claimed dependability goals. In the UK, the Ministry of Defence has dramatically simplified its procurement standards for software. Contractors provide ‘software reliability cases’ justifying the delivered system. Even in the tender stage, a reliability case is required to justify the proposed architecture and to show that the contractor is capable of delivering a case for the development itself [47].

Certifiers and procurers in the US have been slower to adopt direct approaches. Recently, however, a number of government agencies, spearheaded by the High Confidence Software and Systems coordinating group [27], funded a National Academies study on software dependability to address widespread concerns about the costs and effectiveness of the approaches currently used for developing and certifying critical software [32]. This article explains the key elements of a direct approach, drawing heavily on that study.

2 Why Testing Isn’t Good Enough

Testing is a crucial tool in the software developer’s repertoire, and the use of automated tests – especially ‘regression tests’ for catching defects introduced (or reintroduced!) by modifications – is a mark of competence. An awareness of the effectiveness (and relatively low cost) of unit testing has led, in the last few years, to better support in development environments and to a renewed focus on testing in the development process itself.

More extensive and more automated testing can only improve the quality of software, and many researchers are recognizing the potential for harnessing computational resources to increase the power of testing. At the same time, however, despite the famous pronouncement of Edsger Dijkstra that testing can be used to show the presence of errors but not their absence [13], there is a widespread folk belief that extensive testing is sufficient evidence for dependability.

Everything we know about testing indicates that this belief is false. Although some small components can be tested exhaustively, the state space of an entire system is usually so huge that the proportion of scenarios that can be tested is vanishingly small. Software is not continuous, so a successful test for one input says nothing about the system’s response to a distinct but similar input. In practice, it is very hard even to achieve full code coverage – meaning that every statement of the code is executed. Important advances are being made in techniques for generating test suites automatically, and it is possible to subject a program to millions of cases, far more than could ever be constructed by hand. But even the best techniques still fall short; a recent study of a technique that uses constraint solving to force execution down unexecuted branches (and which has proved effective in exposing bugs) was able to cover only 43% of the blocks and 36% of the branches of a .NET component [59].

An alternative approach is to generate tests in a distribution that matches the expected usage profile, adjusted for risk so that most of the testing effort is spent in the most critical areas of functionality. But the number of tests required to obtain high confidence (even with some dubious statistical assumptions) is far larger than one might imagine.

For example, to claim a failure rate of one input in a thousand to a 99% confidence, about 5,000 test cases are needed, assuming no bugs are found [44]. If testing revealed 10 bugs, nearer 20,000 subsequent tests without failure would be needed. Contrary to the intuition of many programmers, finding bugs should not increase confidence that fewer bugs remain but is evidence that there are more bugs to be found.

In summary, while testing may provide adequate confidence that a program is good for a non-critical application, it becomes more and more difficult and expensive as higher and higher levels of assurance are demanded, and cannot deliver the confidence required at a reasonable cost. Most systems will be tested more thoroughly by their users than by their developers, and will often be executing in uncharted territory, exploring combinations of state components that were not tested and perhaps not even considered during design.

There is a particular optimism in the software industry that testing, in the context of a well-structured process, can ensure quality. Perhaps this comes from observing the success of testing in other fields. Many of the dramatic quality improvements in manufacturing that have been achieved in the last century are due in large part to process innovation. Statistical quality control has given us a range of products that fail at low rates and with extraordinary predictability. But unfortunately statistical quality control doesn't work for software. Even within a single company, data about one product may not apply to another (especially a new one), and because software is used in so many ways in so many different environments, even data about a single product may not carry over from one installation to another.

Despite this, most certification regimes still rely primarily on process and testing. Certifiers and developers sometimes talk confidently about achieving 'five nines' of dependability, meaning that the system is expected to survive a hundred thousand commands or hours before failing. The mismatch between such claims and the reality of software failures led one procurer to quip that 'it's amazing how quickly 10^5 hours comes around'.

3 A Direct Approach

The direct approach is straightforward. The desired dependability goal is explicitly articulated as a collection of *claims* that the system has some critical *properties*. An argument, or *dependability case*, is constructed that substantiates the claims. The remainder of this article explains these notions in some more detail, and outlines some of their implications. First we turn to the fundamental question of what constitutes a system, and what it means for it to be dependable.

3.1 What is a System?

The *system* is the engineered product that was introduced to solve a particular problem, consisting of software, the hardware platform on which it runs, the peripheral devices through which it interacts with the environment, and any other components that contribute to achieving its goals – including human operators and users. In many cases, the system's designers must assume that its operators behave in a certain way. An air-traffic management system, for example, cannot prevent a mid-air collision if a pilot is determined to hit an aircraft travelling in the opposite direction on a flight path at an adjacent altitude; eliminating this assumption would require a separation of aircraft that is not economically feasible. When a system's dependability is contingent on assumptions about its operators, they should be viewed as a component of the system, and the design of operating procedures regarded as an essential part of the design effort as a whole.

3.2 What Does 'Dependable' Mean?

A system is *dependable* if can be depended upon – that is trusted or relied upon – to perform a particular task. As I noted early on, such trust or reliance is only rational when evidence of the system's ability to act without producing certain failures has been evaluated. So a system cannot be dependable without evidence, and dependability is thus not merely the absence of defects or the failures that may result from them, but the presence of concrete information suggesting that such failures will not occur.

Dependability is not a binary notion. For greater assurance, better and thus more expensive evidence will be needed. As in all engineering enterprises, dependability is a tradeoff between benefits and risks, with the level of assurance being chosen to match the risk at hand. Our society is not willing to tolerate the failure of a nuclear power station controller, air traffic control center, or energy distribution network, so for such systems we will be willing to absorb larger development and certification costs. Criticality depends, of course, on the context of use. A spreadsheet program becomes critical if it is used for calculating radiotherapy doses. And there are systems – such as GPS satellites and cell phone networks – on which so many applications depend that widespread failure could be catastrophic.

Dependability is not a metric that can be measured on a simple numeric scale, because different kinds of failures have very different consequences. The cost of preventing all failures will usually be prohibitive, so a dependable system will not offer uniform levels of confidence across all functions. On the contrary, a large variance is likely to be a characteristic of a dependable system. Thus a dependable radiotherapy system may become unavailable but cannot be allowed to overdose a patient or log the delivered dose incorrectly; a dependable e-commerce site may display advertisements incorrectly, give bad search results and may perhaps lose shopping cart items over time, but must never bill the wrong amount or leak customers' credit card details; a dependable file synchronizer may report spurious conflicts but should never silently overwrite newer versions of files.

Together, these considerations imply that the first steps in developing a dependable system are to draw the boundary of the system – deciding which components in addition to the software, physical and human, will be relied upon; to identify the critical properties; and to determine what level of confidence is required.

3.3 Properties And Where They Reside

I have talked loosely about a dependable system performing some functions or tasks. For articulating claims about a system's desired behavior, however, even this level of granularity is too coarse. It is preferable instead to focus on critical *properties*. Some will be associated with individual functions, but more often a property will cross-cut several functions.

For dependability, focusing on properties is generally better than focusing on functions, because the properties are what matters. Moreover, they can usually be separated more cleanly from one another, and they retain their meaning as the set of functions offered by the system changes over its lifetime. A critical property of a crime database, for example, may be that every access to the database by a user is logged to some file. Identifying instead some critical subset of logging functions would be inferior, since the full correctness of these functions is likely to be neither necessary nor sufficient for establishing the logging property. Common Criteria, a certification scheme for security, makes this mistake, and focuses attention on the security functions alone, despite the fact that many attacks succeed precisely because they exploit loopholes in other functions that were not thought to be security-related.

Some software systems provide an entirely virtual service but most interact with the physical world. When the purpose of a system is to produce, control or monitor some physical phenomena, these should form the vocabulary for expressing critical properties. This might seem obvious, but there is long tradition of writing requirements in terms of interfaces closer to the software, perhaps because it's easier, or perhaps because of a division of labor that isolates software engineers from system-level concerns. In a radiotherapy application, for example, a critical property is that the patient does not receive an excessive dose, not that the emitted beam has a bounded intensity; nor that the right signal is conveyed to the beam-generating device; nor that the beam settings are computed correctly in the code; and so on.

There is a chain of phenomena connecting the ultimate physical effects of the system at one end through the signals and messages at the peripherals in the middle to the instructions executed in the code at the other end. The more the critical property is formulated using phenomena closer to the software and further from the ultimate effects in the real world domain, the more its correlation to the real concerns of the users is weakened. An infamous accident illustrates the potentially dire consequences of this tendency. An Airbus A320 landing at Warsaw Airport in 1993 was equipped with an interlock that was intended to prevent the pilot from activating reverse thrust while airborne. Unfortunately, the software had been designed to meet a requirement that reverse thrust should be disabled unless wheel pulses were being received (indicating that the wheels were turning and thus in contact with the ground). Due to rain on the runway, the aircraft aquaplaned when it touched down, and the wheels did not turn, so the software dutifully disabled reverse thrust and the aircraft overran the runway. Had the critical property been expressed in terms of being on the ground rather than receiving wheel pulses, the invalid assumption that they were equivalent may have been scrutinized more careful, and the flaw detected. (This account is simplified; for the full incident report see [39]).

This view of requirements is due to Michael Jackson [34] and has been adopted by Praxis in its REVEAL requirements engineering method [25]. More specialized variants of the idea have appeared before, most notably in David Parnas's Four Variable Model [49].

3.4 The Dependability Case

The evidence for dependability takes the form of a *dependability case*. The case is an argument that the software, in concert with other components, establishes the critical properties. What exactly comprises the case – how detailed it will be, what mix of formal and informal arguments is appropriate, and so on – will vary between developments, but certain features are essential.

First, the case should be *auditable* so it can be evaluated by a third-party certifier, independent of both developer and customer. The effort required to check that the case is sound should be much less than the effort required to build the case in the first place. In this respect, a dependability case may be like a formal proof: hard to construct but easy to check. To evaluate a case, a certifier should not need any expert knowledge of the developed system or of the particular application, although it would be reasonable to assume familiarity with the domain area and expertise in software engineering.

Second, the case should be *complete*. This means that the argument that the critical properties hold should contain no holes to be filled by the intuition or expertise of the certifier. Any assumptions that are not justified should be called out, so that it is clear to the certifier who will be responsible for discharging them. For example, the dependability case may assume that a compiler generates code correctly, or that an operating system or middleware platform transports messages reliably, relying on representations by the producers of these components that they provide the required properties; or it might assume that users obey some protocol, relying on the organization that fields the system to train them appropriately. For a product that is not designed with a particular customer in mind, the assumptions become disclaimers: for example, that an infusion pump may fail underwater, or that a file synchronizer will work only if applications do not subvert file modification dates. Assumptions that were made to simplify the case, and which are no more easily substantiated by others, are suspect. Suppose an analysis of a program written in C, for example, declares an assumption that array accesses are within bounds. If this assumption cannot readily be checked, the results of the analysis are suspect and cannot be trusted.

Third, the case should be *sound*. It should not, for example, claim full correctness of a procedure on the basis of non-exhaustive testing; or make unwarranted assumptions that certain components fail independently; or reason, in a program written in a language with a weak memory model, that the value read from a shared variable is the value that was last written to it.

4 Implications

On the face of it, these recommendations – that developers express the critical properties and make an explicit argument that the system satisfies them – are hardly remarkable. If followed, however, they would have profound implications for how software is procured, developed and certified.

4.1 Dependability Case as Product

In theory, one could construct a dependability case *ex post facto*, when the entire development has been completed. In practice, however, this would be near impossible. Constructing the case will be easier and more effective if done hand-in-hand with other development activities, when the rationale for development decisions is fresh and readily available. But there is a far more important reason to consider the dependability case from the very outset of development. By focusing on the case, the developer can make decisions that ease its construction, most notably by designing the system so that critical properties are easier to establish. Decoupling and simplicity, discussed below, offer perhaps the greatest opportunities here.

This is the key respect in which the direct approach to dependability demands a sea change in attitude. Rather than just setting in place some practices or disciplines that are intended to improve dependability, the developers are called upon to consider their decisions in the light of the system's dependability every step of the way, and to view the evidence produced as a work product as integral to the final system as the code itself.

4.2 Procurement

A change to a direct approach affects not only developers but procurers too. The goals set at the start must be realistic – not only for their achievement but also for their demonstration. The FAA specified 3 seconds of downtime per year for the infamous Advanced Automation System, an air-traffic control development that was finally cancelled after several billion dollars was wasted, even though it would have taken 10 years even to obtain the data to substantiate such a claim [9]. The requirement was later revised to 5 minutes.

More fundamentally, however, our society as a whole needs to recognize that the enormous benefits of software inevitably bring risks, and that functionality and dependability are inherently in conflict. If we want more dependable software, we will need to stop evaluating software on the basis of its feature set alone. At the same time, we should be more demanding, and less tolerant of poor quality software. Too often, the users of software have been taught to blame themselves for its failures [48] and to absorb the costs of workarounds themselves.

After the failure of the USS Yorktown's onboard computer system, in which the ship's entire control and navigation network went down after an officer calibrating a fuel valve entered a zero into a database application (in an attempt to overwrite a bad value that the system had produced), blame was initially placed on software [58]. After an investigation, however, the Navy pronounced that human error was in fact the cause. The ship's commanding officer reported that 'Managers are now aware of the problem of entering zero into database fields and are trained to bypass a bad data field and change the value if such a problem were to occur again'. In response to an earlier report that the ship had been towed back to port he noted that the local area network had been restored in two hours. 'It's not something that we desire, but ships do go dead in the water,' he said. 'People sometimes make mistakes and systems break. The trick is we have trained our crew to react to those situations' [57].

With the indirect approach to certification, procurers often give detailed prescriptions for how the software should and should not be developed and which technologies should be used. The direct approach frees the developer to use the best available means to achieve the desired goal; constraints on the development are evaluated by the objective measure of whether they improve dependability or not. Military procurers, for example, have sometimes been wary of object-oriented technology, and have discouraged its use. Viewed from the perspective of constructing a dependability case, a more informed decision can be made, dissecting the elements of object-orientation and their benefits and liabilities. Automatic memory management, for example, might make it easier to reason in a modular way but make establishing time bounds harder; the type system might make it easier, through the use of interfaces, to justify the safety of plugins, but inheritance might compromise encapsulation.

4.3 Structuring Requirements

How requirements are approached sets the tone of the development that follows. A cursory nod to analyzing the problem can result in functionality so unrelated to the users' needs that the developers become mired in endless cycles of refactoring and modification. On the other hand, a massive and windy document can overwhelm the designers with irrelevant details and tie their hands with premature design decisions. Ironically, what a document says can do as much damage as what it fails to say.

In the context of dependability, requirements are even more fundamental. The standard criteria apply: that the developers listen carefully to the stakeholders to understand not merely the functions they say they want, but more deeply the purpose they believe these functions will enable them to accomplish; that the requirements be expressed precisely and succinctly (using, for example, the notions of designations and definitions [33] to clarify terminology); and that great care be taken to avoid making irrevocable decisions when they could be postponed and made later on the basis of much fuller information.

Two criteria take on greater significance, however. First, while prioritizing requirements is important in any system, it matters especially for a critical system. Deciding which requirements are critical (and how critical they are) is the first and most vital design step, determining in large part the cost of the system and the contexts in which it will usable. The architecture of the system will likely be based on these requirements, since (as explained below) the most feasible way to offer high dependability at reasonable cost is to exploit modularity to establish critical properties locally.

Second, in a critical development, it is especially important to record clearly any assumptions that are made about the environment in which the software will operate (including the behavior of human operators). These assumptions will be an integral part of the dependability case, will influence the design, and will become criteria for evaluating in which contexts the software can be deployed.

These two criteria are hardly new but are not always followed. Perhaps under pressure from customers to focus on extensive feature lists, analysts often express requirements as a long catalog of functions. Sometimes the structure follows the sequence of actions performed by a typical user. Functions may be broken into use cases [35], which can be good for describing short, one-off transactions, but which don't address the interactions of multiple parties or end-to-end properties. In a radiotherapy application, for example, the analyst – aware of the risk of delivering incorrect doses, or the risk of unauthorized access – might include sections describing the various functions or use cases associated with logging in, selecting doses, and so on, but might neglect the more important task of describing the critical properties explicitly: that the dose entered by the oncologist should correspond to the dose delivered with the help of the therapist, that access be restricted to certain staff, and so on.

Sometimes developers appreciate the value of prioritization but have been shy to make the bold decisions necessary in downgrading (or eliminating) non-critical requirements. Rather than asking 'what are the critical properties?,' we might ask 'what properties are *not* critical?' If we have trouble answering this latter question, the properties that *are* critical have probably not been identified correctly.

4.4 Decoupling and Simplicity

Building a software system even to relatively low levels of dependability is hard enough, and many large developments collapse under their own weight. How much the cost of a developing a system will increase if some critical properties are to be assured depends on how much of the system is involved. If the critical property is not localized, the entire codebase must be examined to determine whether or not it holds – in short, all the code becomes critical. But if the property can be *localized* to a single component, then attention can be focused on that component alone, and the rest of the codebase can be treated as non-critical. Put another way, the cost of making a system dependable should vary not with the size of the system as a whole but with the extent and complexity of the properties that are assured.

Decoupling is the key to achieving locality. One component is decoupled from another if its behaviour is unaffected by the behaviour of the other. Maximizing decoupling is a guiding principle for software design in general and is fundamental to dependability. It is addressed first during requirements analysis by defining functions and services so that they are self-contained and independent of one another. Then, during design, it is addressed by allocating functionality to components to allow key invariants to be maintained within components and to minimize communication, and by crafting interfaces that do not expose the internals of a service to its clients, or couple clients together unnecessarily. Finally, the decoupling introduced in the design must be realized in the code by using appropriate language features to protect against errors that might compromise it. The USS Yorktown incident discussed above was a failure of decoupling: blame should have been placed not on the operator who entered the bad value but on the designer who made it possible for a single bad entry to bring down an entire networked system.

Amazon's system gives a nice illustration of decoupling. Apparently, the design enforces a rigorous separation between billing and other subsystems, so that more complex (and less dependable) code can be written for less critical features (such as product search) without compromising essential financial properties. Decoupling at the design level will usually require compromises in requirements. Apple's OS X operating system, for example, allows an administrator to change the privileges only of users not currently logged in, and thereby eliminates from the requirements the possibility of privileges being revoked during use, and the complexity and coupling that would follow from it.

Decoupling is one example of the benefits of simplicity in system design. As Tony Hoare famously stated (talking about the design of Ada during his Turing award lecture [28]): '[T]here are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.' Many practitioners are resistant to the claim that simplicity is possible (and some even to the claim that it is desirable). Their doubts are often based on a misunderstanding, either that the advocates of simplicity do not recognize the inherent complexity of the problems solved by computer systems, or that they imagine that simplicity is easily achieved.

Simplicity is not easy to achieve, and – as Alan Perlis noted in one of his famous aphorisms – it tends to follow complexity rather than precede it. The designer of a system that will work in a complex and critical domain faces complex problems. The question is not whether that complexity can be eliminated but whether it can be tamed, so that the resulting system can be as simple as possible under the circumstances. The cost of simplicity may be high, but the cost of lowering the floodgates to complexity is higher. Edsger Dijkstra explained [14]: 'The opportunity for simplification is very encouraging, because in all examples that come to mind the simple and elegant systems tend to be easier and faster to design and get right, more efficient in execution, and much more reliable than the contrived contraptions that have to be debugged into some degree of acceptability.'

4.5 Process and Culture

While process may not be sufficient for dependability, it is certainly necessary. A rigorous process will be needed to ensure that attention is paid to the dependability case, and to preserve the chain of evidence as it is constructed. In the extreme case, if there is no credible process, a certifier has no reason to believe that the deployed software even corresponds to the software that was certified. In a well-known incident in 2003, an electronic voting system was certified for use in an election but a different version of the system was installed in voting booths [62].

A rigorous process need not be a burdensome one. Every engineer involved in a project should be familiar with the process, so it must be described by a brief and easily understood handbook, tailored if necessary to the project at hand. Rather than interfering with the technical work, the process should eliminate a mass of small decisions about how documents are organized, referenced and reviewed, freeing the engineers to concentrate on the more creative aspects of the project. Standards for machine-processable artifacts – especially code – should be designed to maximize opportunities for automation. For example, a coding standard that mandates how variables are named and how specification comments are laid out can make it possible to extract all kinds of cross-referencing and summarization information with lexical tools alone. Bill Griswold has observed that the more the programmer embeds semantic information in the naming conventions of the code, the more readily it can be exploited [23].

One of the paradoxes of software certification is that burdensome processes (such as requiring MCDC test coverage) do seem to be correlated with more dependable software, even though there is little compelling evidence that the processes themselves achieve their stated aims. This may be explained by a social effect. The companies that adhere to the strictest processes tend to attract and reward employees who are meticulous and risk-averse. In the safety critical arena, having a strong 'safety culture' can be the major factor in determining how safe the products are. One study of formal methods found that their success may be more due to the culture surrounding them than to anything more direct [53]. Efforts to build and maintain a strong safety culture can pay dividends. Richard Feynman, in his dissenting report following the Challenger enquiry [17], was effusive in his praise of the constructively adversarial attitude amongst NASA's software engineers, to which he ascribed the high dependability of their software.

Advances in software verification technology may tempt us to imagine (in a Leibnizian fantasy) that one day we will be able to check the dependability of software simply by running it through a machine. But dependability cases will always contain informal elements that cannot be verified mechanically, and whose truth will have to be evaluated not only by an impartial review of the case itself but also by an assessment of the credibility of the organization that produced it. An entirely product-based certification approach thus makes no more sense than one based entirely on process. The organization that produces the software and the software itself are intertwined, and attempts at improving dependability, and efforts to measure it, must take both into account.

4.6 Robust Foundations

A skyscraper cannot be easily built on sand, and likewise a robust software system cannot be built on a foundation of weak tools and platforms. Fifty years after the invention of static typing and automatic memory management, the decision to use an unsafe programming language such as C or C++ (that provide neither) requires serious justification, and for a critical system the benefits that are obtained in compensation for the loss of safety will have to be extraordinarily compelling. Arguments against safety based on performance are usually overstated. And with Java and C# now widely known and available, and equipped with impressive libraries, there is no longer a reason to consider safe languages as boutique technologies.

The value of static typing is often misunderstood. It is not that type errors don't occur during execution. They do – because most statically typed languages are sufficiently complex that some checks are inevitably postponed to runtime. Similarly, the value of strong typing is not that runtime type errors are more acceptable than other kinds of failure. An unanticipated failure is never good news. Moreover, runtime type checking can make things worse: the Ariane 5 rocket might not have been lost had an arithmetic overflow in an irrelevant module not been propagated to the top level.

Strong typing has two primary benefits. First, it prevents a module from writing to regions of memory that it cannot name (through local and global variables and sequences of field accesses). This means that a syntactic dependence analysis can determine the potential couplings between modules, and can be used to establish that one module is decoupled from another, thus making it possible to ignore the latter when analyzing the behaviour of the former. Second, strong typing makes runtime failures happen earlier, as soon as a type error occurs, rather than later when the failure is likely to be harder to diagnose and may have done more damage. Static typing provides the important additional advantage of catching many type errors at compile time. This is extremely valuable, because type errors are often symptoms of serious errors and structural flaws.

Complexity in a programming language can compromise dependability, because it increases the chance of misunderstandings in which the program does not have the behavior that the programmer envisaged. It is important to avoid obscure mechanisms, especially those that have a platform-dependent interpretation. Coding standards can be very helpful in taming dangerous language features [26, 29].

Progress in language design has produced major improvements but old lessons are easily forgotten. The original design of Java, for example, lacked iterators (as a control construct) and generics, and did not unify primitive types and objects – all features of CLU in 1975 [43]. When it was realized that these features were essential, it was too late to incorporate them cleanly. Others have argued that the design of Java's concurrency constructs ignored 25 years of research in parallel programming [7].

It may be curmudgeonly to complain about Java, especially since it brings so many good ideas to mainstream programming – especially strong typing – that might otherwise have languished in obscurity. And, to be fair, Java incorporates features of older languages in a more complex setting; subtyping, in particular, makes it much harder to incorporate other features (such as generics). Nevertheless, it does seem sad that the languages adopted by industry often lack the robustness and clarity of their academic predecessors.

It is important to recognize that dependability was not the primary goal in the design of most programming languages. Java was designed for platform independence, and its virtual machines include a class loader that absorbs much of the complexity of installation variability. As a result, however, a simple call to a constructor in the source code sets in motion a formidable amount of machinery that could compromise the system's dependability.

The choice of computing platform – operating system, middleware, database, etc – must also be carefully considered. A platform that has been widely adopted for general applications usually has the advantage of lower cost and a larger pool of candidate developers. But commodity platforms are not usually designed for critical applications, so when high dependability is required, enthusiasm for their use should be tempered with the risks involved.

4.7 The (In)Significance of Code

A dependability argument is a chain with multiple links. One link may argue that a software component has some property; another that a peripheral behaves in a certain way; yet another that a human operator obeys some protocol; and together, these might establish the end-to-end dependability requirement.

The argument is only as strong as the weakest link in the chain. Many software engineers and researchers are surprised to learn that the correctness of the code is rarely the weakest link. In an analysis of fatal accidents that were attributed to software problems, Donald MacKenzie found that coding errors were cited as causes only 3% of the time [45]. Problems with requirements and usability dwarf the problems of bugs in code, suggesting that the emphasis on coding practices and tools in both research and industry may be mistaken. Exploiting tools to check arguments at the design and requirements level may be more important, and is often more feasible since artifacts at the higher level are much smaller [31].

Nevertheless, the correctness of code is a vital link in the dependability chain. Even if the low incidence of failures due to bugs reflects success in improving code quality, the cost is still unacceptable [19], especially when very high assurance is required. And in the arena of security rather than safety, code vulnerabilities are responsible for a much higher proportion of failures.

4.8 Testing and Analysis

Testing is a crucial part of any software development process, and its effectiveness can be amplified by liberal use of runtime assertions (especially those checking complex invariants), by formulating tests early on, by creating tests in response to bug reports, and by integrating testing into the build so that tests are run frequently and automatically. But, as discussed above, testing cannot generally deliver the high levels of confidence that are required for critical systems, and analysis is needed to fill the gap.

Analysis might involve any of a variety of tools and techniques, according to the kind of property being checked and the level of confidence required. In the last decade, dramatic advances have been made in a variety of analyses that establish properties of code fully automatically, using theorem proving [eg, 18], static analysis [eg, 40], model checking [eg, 61] and model finding [eg, 12].

How well these techniques will work, and how widely they will be adopted, remains to be seen. But a number of industrial successes demonstrate that the approaches are at least feasible and effective in the right context. Microsoft, for example, now includes a sophisticated verification component in its driver development toolkit [3]; Praxis has achieved extraordinarily low defect rates using a variety of formal methods [24]; and Airbus has used static analysis to show the absence of low-level runtime errors in the A340 and A380 flight control software [11].

Until these approaches are more widely adopted, many development teams will choose to rely instead on manual code review. It is important to realize that arguments that are not mechanically checked are likely to be flawed, so their credibility must suffer, and the confidence of any dependability claims that rely on them must be reduced accordingly.

4.9 The Credibility of Tools

Tools are enormously valuable, but the glamour of automation can sometimes overwhelm our better judgment. A symptom of this is our tendency to invest terms used to describe tools with more significance than their simple meaning. From early on, for example, program analyses have been classified by their inventors as 'sound' or 'unsound'. A sound analysis establishes a property with perfect reliability. That is, if the analysis does not report a bug, then there is no possible execution that can violate the property. This notion helpfully distinguishes verifiers from bug finders [eg, 15] – a class of tools that are very effective at catching defects, especially in low quality software, but which usually cannot contribute evidence of dependability, since they tend to be heuristic and therefore unsound.

But the assumption that sound tools are inherently more credible is dangerous. Alex Aiken found that an unsound tool uncovered errors in a codebase that a prior analysis, using a sound tool, had failed to catch. The much higher volume of false alarms produced by the sound tool overwhelmed its users and made the real defects harder to identify [1]. In recent years, developers of analysis tools have come to realize that the inclusion of false positives is just as problematic as the exclusion of true positives and that more sophisticated measures are needed – such as the notions of precision and accuracy from the field of information retrieval.

Even if an analysis establishes a property with complete assurance, the question still remains of whether the property itself is sufficient. Eliminating arithmetic overflows and array bounds errors from a program, for example, is certainly progress, but knowing that such faults are absent may not help the dependability case unless there is either a chain of reasoning connecting this knowledge to assertions about end-to-end properties, or some strong statistical evidence that the absence of these faults is correlated with the absence of other faults.

Amongst analysis tools, mathematical proof is generally believed to offer the highest level of confidence. A proof can usually be checked relatively easily, so that an analysis substantiated with a proof can be certified independently by examining the proof in isolation, mitigating the concern that the tool that produced the proof might have been faulty.

Proof is not foolproof, however. Joshua Bloch found [6] that the binary search algorithm that Jon Bentley had proven correct in his column on program verification [4] harbored a subtle flaw, and that the very same bug appeared in Bloch's own implementation of the algorithm in the Sun Java library. The problem arose when the sum of the low and high bounds exceeded the largest representable integer. Of course the proof wasn't wrong in a technical sense; there was an implicit assumption that no integer overflow would occur (which was reasonable when Bentley wrote his article, since computer memories back then were not large enough to hold such a large array). In practice, such assumptions will always pose a risk, since they are hidden in the very tools we use to reason about systems, and we may not be aware of them until they are exposed.

Assumptions create another problem for proofs. An inadvertent contradiction amongst the assumptions makes the proof worthless, since anything can be inferred from it – including the property the user was hoping to establish. Code verification tools typically feed to the theorem prover not only the formulas extracted from the program but additionally a library of axioms. Any unintentional overconstraint in these axioms may cause properties to be inferred from the code that do not in fact hold. The same problem arises in software design analysis and model checking, where coverage techniques have been developed to highlight portions of the design that were unused in establishing the result [see, eg, 60, 38].

5 Closing Thoughts

Software plays a fundamental role in our society, bringing enormous benefits in all fields. Because the power of software is irresistible, we depend on it increasingly. Many of our current systems are highly centralized and tightly coupled [50] and are thus vulnerable to massive, coordinated failure.

A Chicago hospital lost its entire pharmacy database one night, and was only able to reconstruct medication records for its patients by collecting paper printouts from nurses' stations. In their report on this incident [10], Richard Cook and Michael O'Connor concluded: 'Accidents are signals sent from deep within the system about the sorts of vulnerability and potential for disaster that lie within.' Similar signals have been sent by electronic voting [36], air traffic control [21], nuclear power [37] and energy distribution [51] systems.

The growing tendency to embed software in powerful and invasive physical devices brings greater risk, especially in medicine where software can save lives but also kill [54]. Software problems led to the recall of 200,000 implanted pacemakers and defibrillators between 1990 and 2000 [46]. In the 20 years prior to 2005 the FDA recorded 30,000 deaths and 600,000 injuries from medical device failures [16]. How many of these can be attributed to software is unclear, although separate studies have found that about 8 percent of medical device recalls are software related. Moreover, few of the device failures that occur – perhaps only 1 in 40 – are actually reported [20], so the actual incidence of injuries may be higher.

As software becomes a more and more critical component of our civic infrastructure, the question of its dependability will demand an answer. Currently we have little data on how often our software fails, so as a society we are unable to make rational decisions about how to balance the risks of new systems against the benefits they bring, and about how to allocate resources to development and certification. Our society could benefit greatly from a systematic effort to collect and analyze software failure data – perhaps even a kind of NTSB for software.

Our assumptions about the dependability of existing systems are often not based on hard evidence. Many in the certification community are already skeptical of the assumed link between certification at particular 'safety integrity levels' and failure rates. Level 3 of IEC-61508, for example, supposedly offers 10^7-10^8 hours of continuous mode operation without error, even though there is little evidence that systems accredited to that level ever achieve such low failure rates. One consequence of adopting a more scientific approach to dependability will undoubtedly be a reduction in the claims made for existing systems, and perhaps a recognition that lower levels of dependability suffice. On the other hand, we may find that some systems – GPS navigation devices, for example – do not warrant the reliance that many users place on them.

The message of this article has been this: that it is not rational to believe that a software system is dependable without good reason. Any approach that promises to develop dependable software must produce such reason. A clear and explicit articulation is needed of what 'dependable' means for the system at hand, and an argument needs to be given that takes into account not only the correctness of the code, but also the behaviour of all the other components of the system, including human operators.

Is this approach practical? The cost of constructing a dependability case may be high. On the other hand, an awareness, from the very start of the development, of the goal of constructing the dependability case, should focus resources where they bring the greatest return, and the effort invested in obtaining a decoupled design may reduce the cost of maintenance later. The experience of Praxis shows that many of the approaches that the industry regards as too costly (such as formal specification and static analysis of code) can actually reduce overall cost [2].

Augmenting testing with more ambitious analysis tools will require greater expertise than is available to many teams today but may not increase the cost. When low levels of confidence suffice, testing may be the most cost-effective way to establish dependability. As the level of confidence rises, though, testing soon becomes prohibitively expensive, and techniques that make better use of computational resources are likely to be more economical. Invariants may be harder to write than test cases, but a single invariant defines an infinite number of test cases, so a decision to write one (and use a tool that checks all the cases it defines) will pay off very soon.

Efforts to make software more dependable or secure are inherently conservative and therefore risk retarding progress, and many practitioners understandably see certification schemes and standards as a millstone around their neck. A direct approach based on dependability cases gives developers an incentive to use whatever development methods and tools are most economic and effective, and therefore rewards innovation.

If we had reason to believe that our systems should not fail, then, when they did, we would be able to analyze their failures and learn from the experience [52]. Accounts of failures in civil engineering make edifying reading in part because they often refute in subtle respects the theories on which the designs were based. Mario Salvadori preceded *Why Buildings Fall Down* [42] with *Why Buildings Stand Up* [56]. With a flaw in a dependability argument to point to, such a book might one day be written about software, and instead of constantly complaining about the fragility and unpredictability of software, we will marvel at how dependable it is, and how rare and subtle are its failures.

Acknowledgments

The key ideas in this article come from a National Academies study [32] that I chaired. I am very grateful to the members of my committee – Joshua Bloch, Michael DeWalt, Reed Gardner, Peter Lee, Steven Lipner, Charles Perrow, Jon Pincus, John Rushby, Lui Sha, Martyn Thomas, Scott Wallsten and David Woods – and to our study director Lynnette Millett for their contributions, to our sponsors (especially Helen Gill who was instrumental in making the case for the study) and to Jon Eisenberg, director of the CSTB, who was a supportive advocate of our work. I am especially grateful to Martyn Thomas who articulated the direct approach from early on and shared the editing of the study report. Many of the opinions expressed in this article, however, are my own and have not been approved by the committee or by the Academies.

Thank you to Butler Lampson, Shari Lawrence Pfleeger and Derek Rayside who gave very extensive and helpful suggestions on an initial draft of the article; to the anonymous reviewers; and to Hari Balakrishnan, Joshua Bloch, Peter Lee, Steve Lipner, Bill Maisel, Lynnette Millett, Andrew Myers, Charles Perrow, Martyn Thomas and Scott Wallsten who gave valuable feedback and shared their expertise on particular topics.

References

[1] Y. Xie and A. Aiken. Context- and Path-Sensitive Memory Leak Detection. *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2005, pp.115–125.

[2] Peter Amey. Correctness by Construction: Better Can Also Be Cheaper. *CrossTalk Magazine*, The Journal of Defence Software Engineering, March 2002. Available online at http:// www.praxis-his.com/pdfs/c_by_c_better_cheaper.pdf.

[3] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *29th ACM Symp. on Principles of Programming Languages* (POPL '02), 2002, pp. 1–3.

[4] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986; Second Edition, 2000)

[5] Peter Bishop, Robin Bloomfield and Sofia Guerra. The Future of Goal-Based Assurance Cases. Supplemental Volume of the *2004 International Conference on Dependable Systems and Networks*, Florence, Italy, June 2004, pp. 390–395.

[6] Joshua Bloch. Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.

[7] Per Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, Volume 34, Number 4, 1999, pp.38–45.

[8] Joel Brinkley. Out of Spotlight, Bush Overhauls U.S. Regulations. *New York Times*, August 14, 2004.

[9] Edward Cone. The Ugly History of Tool Development at the FAA. *Baseline Magazine*, 2002-04-09.

[10] Richard Cook and Michael O'Connor. Thinking about accidents and systems. In: H.R. Manasse and K.K. Thompson (eds.), *Medication Safety: A Guide to Health Care Facilities*.

American Society of Health-System Pharmacists, Washington, DC, pp. 73–87. Available online at http:// www.ctlab.org/documents/ASHP_chapter.pdf.

[11] Patrick Cousot. Proving the Absence of Run-Time Errors in Safety-Critical Avionics Code. In *Proceedings of the Seventh ACM & IEEE International Conference on Embedded Software* (EMSOFT 2007), Salzburg, Austria, September 2007, ACM Press, pp. 7–9.

[12] Greg Dennis, Felix Sheng-Ho Chang and Daniel Jackson. Modular Verification of Code with SAT. *International Symposium on Software Testing and Analysis*, Portland, ME, July 2006.

[13] Edsger W. Dijkstra. *On the Reliability of Programs*. Available at: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF

[14] Edsger W. Dijkstra. The Tide, Not the Waves. In: *Beyond Calculation: The Next Fifty Years of Computing*, Peter J. Denning and Robert M. Metcalfe (eds.), Copernicus (Springer-Verlag), 1997, pp. 59–64.

[15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[16] FDA. *Ensuring the Safety of Marketed Medical Devices: CDRH's Medical Device Postmarket Safety Program*. January 2006.

[17] Richard P. Feynman. Appendix F–Personal Observations on the Reliability of the Shuttle. In *Report of the Presidential Commission on the Space Shuttle Challenger Accident*, June 1986. Available at http://science.ksc.nasa. gov/shuttle/missions/51-l/docs/rogers-commission/Appendix-F.txt.

[18] C. Flanagan, K. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended Static checking for Java. In *ACM Conference on Programming Language Design and Implementation* (PLDI '02), 2002, pp. 234–245.

[19] M.P. Gallaher and B.M. Kropp. *Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology, May 2002.

[20] GAO, 1986. *Medical Devices: Early Warning of Problems is Hampered by Severe Underreporting*. US Government Printing Office, Washington, DC, GAO publication PEMD-87-1.

[21] Linda Geppert. Lost Radio Contact Leaves Pilots On Their Own. *IEEE Spectrum*, November 2004. Available online at: http://www.spectrum.ieee.org/nov04/4015.

[22] Andy German and Gavin Mooney. Air Vehicle Software Static Code Analysis – Lessons Learnt. *Proceedings of the Ninth Safety-Critical Systems Symposium*, Felix Redmill and Tom Anderson, eds., Springer-Verlag, Bristol, United Kingdom, 2001.

[23] W. G. Griswold. Coping with Crosscutting Software Changes Using Information Transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, September 2001.

[24] Anthony Hall. Using formal methods to Develop an ATC Information System. *IEEE Software*, 13(2):66-76, 1996.

[25] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *Proceedings of the 5th International Symposium on Requirements Engineering*, August 2001.

[26] Les Hatton. *Safer C: Developing Software in High-Integrity and Safety-Critical Systems*. McGraw-Hill, 1995.

[27] High Confidence Software and Systems Coordinating Group, National Coordination Office (NCO) for Networking and Information Technology Research and Development (NITRD). http://www.nitrd.gov/subcommittee/hcss.php.

[28] C.A.R. Hoare. The Emperor's Old Clothes. *Communications of the ACM*, Volume 24, No. 2, 1981, pp. 75–83.

[29] G.J. Holzmann. The Power of Ten: Rules for Developing Safety Critical Code. *IEEE Computer*, 39(6):95–97, 2006.

[30] IAEA, *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama: Report of a Team of Experts*, 26 May – 1 June 2001, IAEA (Vienna, Austria). Available online at: http://www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf

[31] Daniel Jackson. Dependable Software by Design. *Scientific American*. June 2006. Available at: http://www.sciam.com/article.cfm?id=dependable-software-by-de&collD=1

[32] Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds. *Software For Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences, National Research Council of the National Academies. The National Academies Press, Washington, DC. 2007. Available at: http://books.nap.edu/openbook.php?isbn=0309103940.

[33] Michael Jackson. *Software Requirements and Specifications*, Addison-Wesley and ACM Press, 1996.

[34] Michael Jackson. Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley, Boston, Massachusetts, 2001.

[35] Ivar Jacobson. *Object-Oriented Software Engineering*, Addison Wesley Professional, 1992.

[36] Grant Gross. E-Voting Vendor: Programming Errors Caused Dropped Votes. *Network World*, August 22, 2008. Available online at: http://www.networkworld.com/news/2008/082208-e-voting-vendor-programming-errors-caused.html

[37] Brian Krebs. Cyber Incident Blamed for Nuclear Power Plant Shutdown. *Washington Post*. June 5, 2008. Available online at: http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958_pf.html

[38] Orna Kupferman and Moshe Y. Vardi. Vacuity Detection in Temporal Model Checking. *Software Tools for Technology Transfer* 4(2):224-233, 2003.

[39] Peter Ladkin, transcriber. *Transcription of Report on the Accident of Airbus A320-211 Aircraft In Warsaw on 14 September 1993*. Main Commission, Aircraft Accident Investigation, Warsaw. Available at http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html.

[40] T. Lev-Ami, R. Manevich, and M. Sagiv. TVLA: A System for Generating Abstract Interpreters. In P. Jacquart, editor, *Building the Information Society*, Chapter 4, pp. 367–376. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[41] Nancy Leveson and Clark Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26:18-41, July 1993.

[42] Matthys Levy and Mario Salvadori. *Why Buildings Fall Down: How Structures Fail*. Norton, 1992.

[43] Barbara Liskov. A History of CLU. *History of Programming Languages Archive*, Second ACM SIGPLAN Conference on History of Programming Languages, 1993, pp. 133–147.

[44] Bev Littlewood and David Wright. Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software. *IEEE Transactions On Software Engineering*, Vol. 23, No. 11, November 1997, p. 673.

[45] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, Cambridge, Massachusetts, 2001. Chapter 9.

[46] William H. Maisel, Michael O. Sweeney, William G. Stevenson, Kristin E. Ellison, Laurence M. Epstein. Recalls and safety alerts involving pacemakers and implantable cardioverter- defibrillator generators. *Journal of the American Medical Association*, 2001, 286:793–799.

[47] Ministry of Defence. *Defence Standard 00-42: Reliability And Maintainability Assurance Guides, Part 2: Software*. Issue 1, September 1997.

[48] Donald Norman. *The Design of Everyday Things*. MIT Press, August 1998.

[49] D.L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 1995, 25(1):41–61.

[50] Charles Perrow. *Normal Accidents*, Princeton University Press, 1999.

[51] Charles Perrow, *The Next Catastrophe: Reducing our Vulnerabilities to Natural, Industrial, and Terrorist Disasters*. Princeton University Press, 2004. Chapter 7.

[52] Henry Petroski. *To Engineer Is Human: The Role of Failure in Successful Design*. Vintage Books, 1985.

[53] Shari Lawrence Pfleeger and Les Hatton. Investigating the Influence of Formal Methods. *Computer*, Volume 30, Number 2, 1997, pp. 33–43.

[54] Jonathan Rockoff. Flaws In Medical Coding Can Kill: Spread Of Computers Creates New Dangers, FDA Officials Warn. *Baltimore Sun*. June 30, 2008. Available online at: http://www.baltimoresun.com/news/health/bal-te.fda30jun30,0,912831.story.

[55] Marc Ross and Tom Wenzel. Are SUVs Safer than Cars? An Analysis of Risk by Vehicle Type and Model. *Transportation Research Board 82nd Annual Meeting*, Washington DC, January 15, 2003. Available at: http://eetd.lbl.gov/ea/teepa/pdf/TRB_Safety_1-03.pdf

[56] Mario Salvadori. *Why Buildings Stand Up: The Strength of Architecture*. Norton, 1980.

[57] Gregory Slabodkin. Navy: Calibration flaw crashed Yorktown LAN. *Government Computing News*, November 9, 1998. Available at: http://www.gcn.com/print/17_30/33914-1.html.

[58] Gregory Slabodkin. Software Glitches Leave Navy Smart Ship Dead In The Water. *Government Computer News*, July 13, 1998. Available online at http://www.gcn. com/print/17_17/33727-1.html.

[59] Nikolai Tillmann and Jonathan de Halleux. Pex – White Box Test Generation for .NET. *Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008)*, LNCS, vol. 4966, pages 134-153, April 2008.

[60] Emina Torlak, Felix Sheng-Ho Chang and Daniel Jackson. Finding Minimal Unsatisfiable Cores of Declarative Specifications. *15th International Symposium on Formal Methods* (FM'08). Turku, Finland. May 2008.

[61] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, Volume 10, Number 2, April 2003.

[62] Kim Zetter. E-Voting Undermined by Sloppiness. *Wired*, December 2003. Available online at: http://www.wired.com/politics/security/news/2003/12/61637.