DOI:10.1145/1498765.1498787

Who could fault an approach that offers greater credibility at reduced cost?

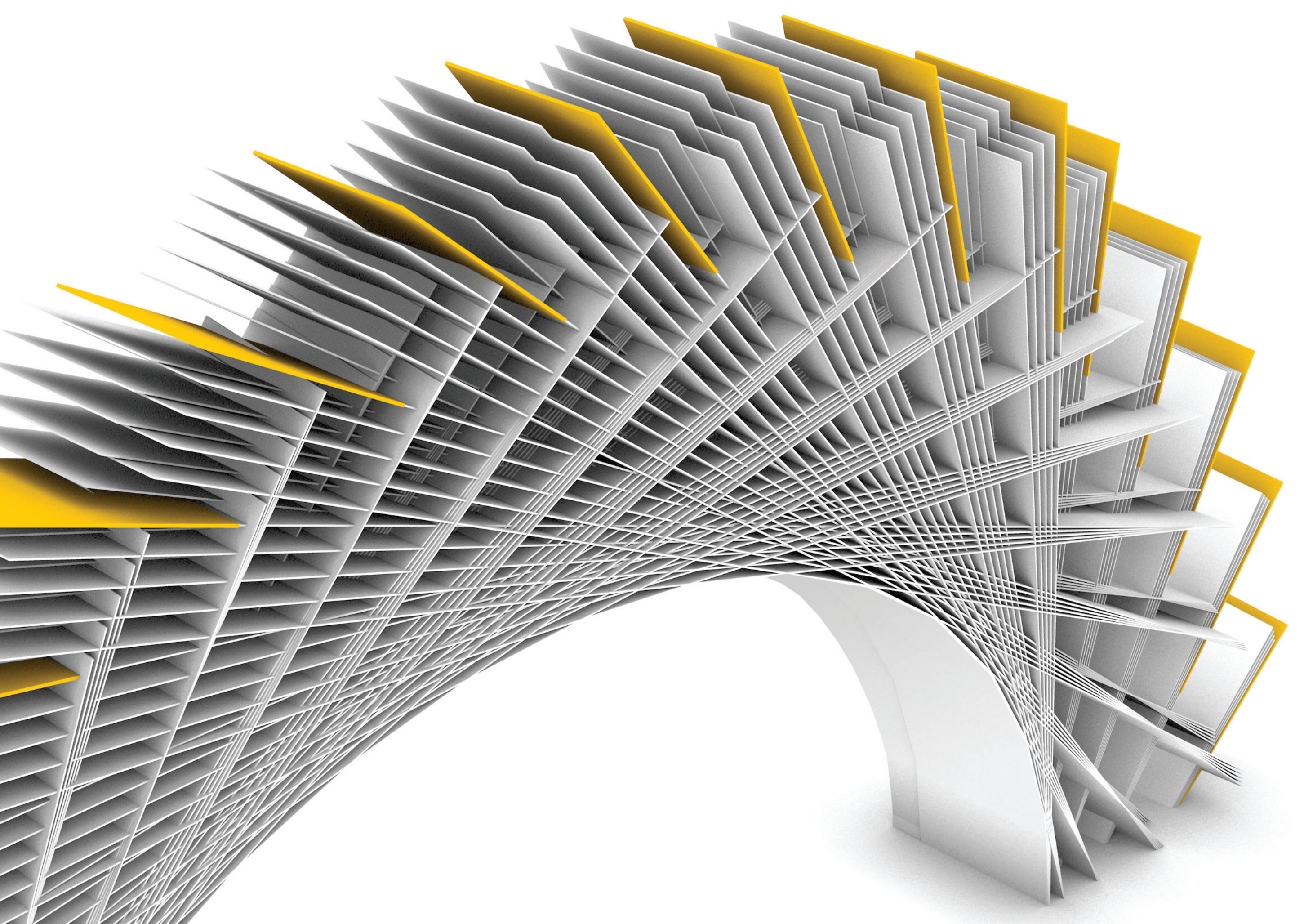
BY DANIEL JACKSON

A Direct Path to Dependable Software

SOFTWARE PLAYS A fundamental role in our society, bringing enormous benefits to all fields. But because many of our current systems are highly centralized and tightly coupled,[33] we are also susceptible to massive and coordinated failure.

A Chicago hospital lost its entire pharmacy database one night, and it was only able to reconstruct medication records for its patients by collecting paper printouts from nurses' stations. In their report on this incident,[6] 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 asent, for example, in the fields of electronic voting,[24] air traffic control,[13] nuclear power,[25] and energy distribution.[34]

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.[36] Software problems led to the recall of 200,000 implanted pacemakers and defibrillators between 1990 and 2000.[30] In the 20 years prior to 2005, the U.S. Food and Drug Administration (FDA) recorded 30,000 deaths and 600,000 injuries from medical-device failures.[9] How many of these incidents can be attributed to software is unclear, though separate studies have found that about 8% of medical-device recalls are software-related. Moreover, few of the device failures that occur—perhaps only 1 in 40—are actually reported,[12] so the actual incidence of injuries is likely to be higher.

What would it take to make software more dependable? Until now, most approaches have been indirect, involving practices—processes, tools, or techniques—believed to yield dependable software. The case for dependability has thus rested on the extent to which the developers adhered to these practices. This article argues that developers should instead produce *direct* evidence of their software's dependability. The potential advantages of this approach are greater credibility (as the claim is not contingent on the effectiveness of the practices) and reduced cost (because development resources can be focused where they have the most impact).

The Need for a Direct Approach

A dependable system is one you can *depend* on—that is, you can place your trust in it. A rational person or organization only does this with *evidence* that the system's benefits far outweigh its risks. Without such evidence, a system cannot be depended on, in much the same way that a download from an unknown Web site 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 resulting software's quality. Today, however, we are far from that goal. Although individual companies can predict defect rates within product families based on historical data, industrywide data collection and analysis barely exist.

Contrast software systems with cars, for example. In the U.S., 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. NHTSA also receives data from auto companies regarding warranty claims, defects, and customer complaints. Similarly, the National Transportation Safety Board (NTSB), best known for its work in aviation, analyzes highway accidents and issues reports on, among 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 therefore cannot benefit from industry data that would improve their designs and development strategies, and consumers cannot use such

ILLUSTRATION BY MIKAEL CHRISTENSEN

data to make informed purchasing decisions.

Actually, where data is collected, it is often suppressed; many companies withhold even basic information about the number and severity of defects in their products, even when issuing patches that purport to resolve them. And no government agency is charged with investigating software failures or even recording software-related fatal accidents. When an accident report does implicate software, it rarely includes enough information to allow any general lessons to be learned.

Over the past few decades we have developed approaches and technologies that can dramatically improve the quality of software. They include better platforms (safe programming languages, operating systems with address-space separation, virtual machines), better development infrastructure (configuration control, bug tracking, traceability), better processes (spiral and agile models, prototyping), and better tools (integrated environments, static analyzers, model checkers). Moreover, we have made progress in understanding the fundamentals, for example, of problem structuring, design modeling, software architecture, verification, and testing. 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 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 they have had 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, resulting in voluminous documentation of questionable value. The Common Criteria security certification that Microsoft obtains for its operating systems, for example, costs more than its internally devised mitigations but is believed by the company to be far less 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, however arbitrary, was adhered to and some amount of testing, whether conclusive or not, was performed. Not surprisingly, certified systems sometimes fail catastrophically. A particularly tragic example was the failure of an FDA-certified radiation-therapy machine in Panama in 2001;[20] fatal overdoses resulted from poorly engineered software in an incident reminiscent of the Therac failures of 15 years earlier. Even the most highly regarded standards demand expensive practices whose value is hard to assess. DO178B, for example, the safety standard used in the U.S. for avionics systems, requires a level of test coverage known as MCDC that is extremely costly and whose benefits studies have yet to substantiate.[14]

A very different approach, sometimes called "goal-based" or "case-based" certification, is now gaining currency. Instead of particular practices being mandated, the developer is instead called upon to provide *direct* evidence that the particular system satisfies its claimed dependability goals. In the U.K., the Ministry of Defence has dramatically simplified its procurement standards for software under this approach, with contractors providing "software reliability cases" to justify the system. Even in the early stages, a reliability case is required to defend the proposed architecture and to show that the contractor is capable of making a case for the development itself.[31]

This direct approach has not yet been adopted by American certifiers and procurers. Recently, however, a number of government agencies, spearheaded by the High Confidence Software and Systems Coordinating Group, funded a National Academies study to address widespread concerns about the costs and effectiveness of existing approaches to software dependability.[22] The direct approach recommended by the study is the basis of this article.

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 by modifications—is a mark of competence. More extensive 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 yet further. 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, there is a widespread folk belief that 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 executed in a typical test is vanishingly small. Software is not continuous, so a successful test for one input says nothing about the system's response to a similar but distinct input. In practice, it is very difficult even to achieve full code coverage—that is, with every statement of the code being executed.

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.[28] If testing reveals 10 bugs, nearer to 20,000 subsequent tests without failure are needed. Contrary to the intuition of many programmers, finding bugs should not increase confidence that fewer bugs remain; indeed, it is evidence that there are more bugs to be found.

Thus while testing may provide adequate confidence that a program is good for a noncritical application, it becomes increasingly difficult and expensive as higher levels of assurance are demanded; under such circumstances, testing 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.

Nevertheless, most certification regimes still rely primarily on testing. Developers and certifiers sometimes talk self-assuredly about achieving "five nines" of dependability, meaning that the system is expected to survive 100,000 commands or hours before failing. The mismatch between such claims and the reality of software failures led one procurer to quip "It's amazing how quickly 10^5 hours comes around."

A Direct Approach

The direct approach, by definition, 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 develops these notions and outlines some of their implications, but first we turn to the fundamental questions of what constitutes a system and what it means for the system to be dependable.

What is a system? An engineered product that is introduced to solve a particular problem and that consists of software, the hardware platform on which the software runs, the peripheral devices through which the product interacts with the environment, and any other components that contribute to achieving the product's goals (including human operators and users) is considered a *system*. 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 midair collision if a pilot is determined to hit another aircraft; eliminating this assumption would require a separation of aircraft that would not be 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 overall design.

What does "dependable" mean? A system is *dependable* if can be depended on—that is, trusted—to perform a particular task. As noted earlier, such trust is only rational when evidence of the system's ability to act without exhibiting certain failures has been assessed. 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.

As in all engineering enterprises, dependability is a trade-off between benefits and risks, with the level of assurance (and the quality and cost of the evidence) being chosen to match the risk at hand.

As in all engineering enterprises, dependability is a trade-off between benefits and risks, with the level of assurance (and the quality and cost of the evidence) being chosen to match the risk at hand. Our society is not willing to tolerate the failure of a nuclear power plant, 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, say, for calculating radiotherapy doses. And there are systems, such as GPS satellites and cellphone 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. In fact, 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; a dependable e-commerce site may display advertisements incorrectly, give bad search results, and perhaps lose shopping-cart items over time, but it 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 involve drawing its boundaries—deciding which components in addition to the software, physical and human, will be relied on; identifying the critical properties; and determining what level of confidence is required.

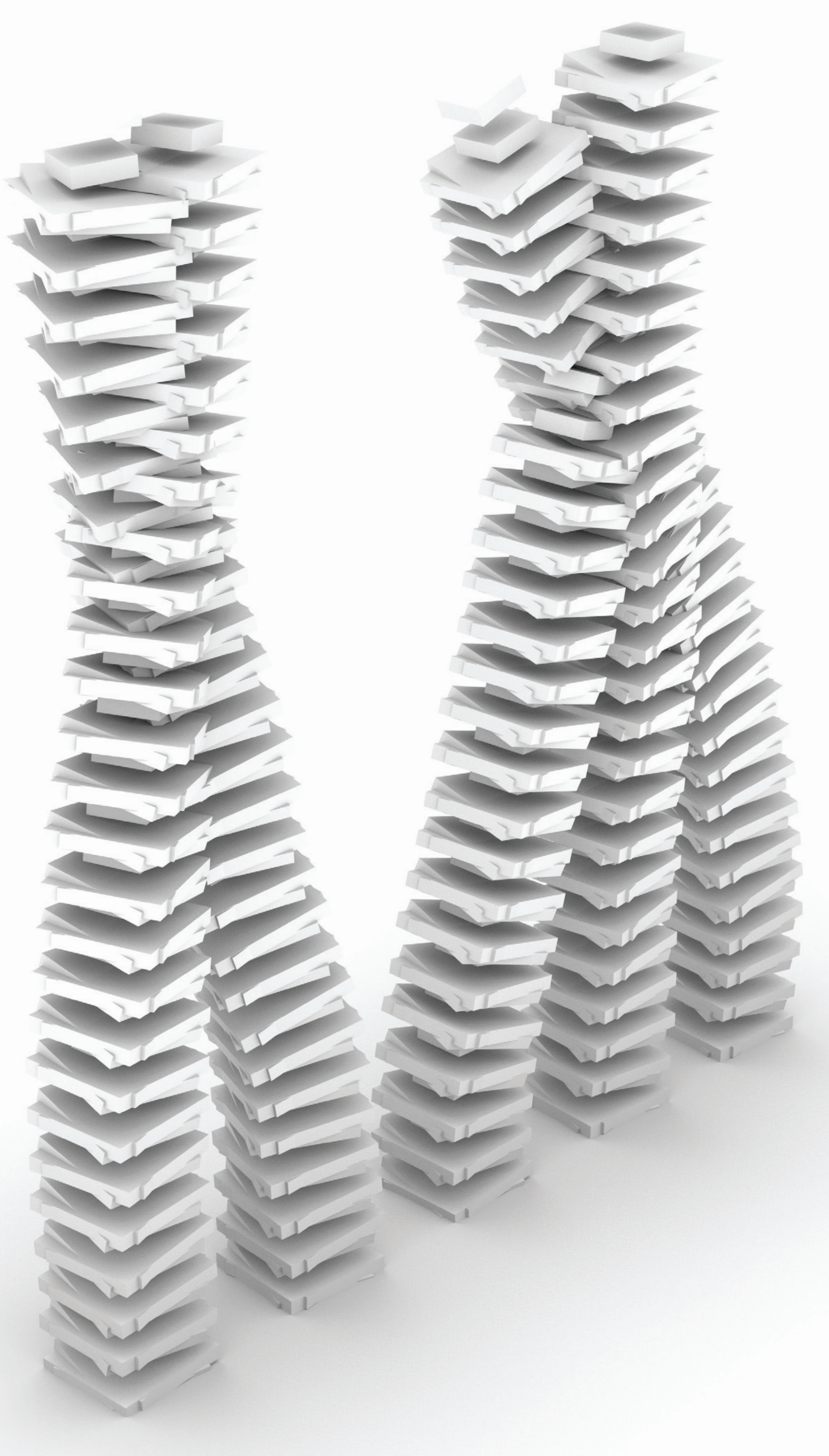


ILLUSTRATION BY MIKAEL CHRISTENSEN

Properties and where they reside. So far I have talked loosely about a dependable system performing some functions or tasks. But for articulating claims about a system's desired behavior, 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 crosscut several functions.

For dependability, focusing on properties is generally better than focusing on functions because the properties are what matter. 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 in some file. Identifying instead some critical subset of logging functions would be inferior, as the full correctness of these functions would likely be neither necessary nor sufficient for establishing the logging property. Common Criteria, a certification scheme for security, makes this mistake; it 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 particular physical phenomena, they 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 because of a division of labor that isolates software engineers from system-level concerns. In a radiotherapy application, for example, a critical property is not that the emitted beam has a bounded intensity, or that the right signal is conveyed to the beam-generating device, or that the beam settings are computed correctly in the code. It is that the patient does not receive an excessive dose.

There is a chain of events connecting the ultimate physical effects of the system at one end back through the signals of 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 away from the ultimate effects in the real world, the more its correlation to the fundamental concerns of the users is weakened.

An infamous accident illustrates the potentially dire consequences of this too-close-to-the-software tendency. An Airbus A320 landing at Warsaw Airport in 1993 was equipped with an interlock intended to prevent the pilot from activating reverse thrust while airborne.

Unfortunately, the software had been designed to meet a requirement that reverse thrust be disabled unless wheel pulses were being received (indicating that the wheels were turning and thus in contact with the ground). Because of 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 carefully and the flaw detected. (This account is simplified; for the full incident report see Ladkin[26]).

This view of requirements is due to Michael Jackson[23] and has been adopted by Praxis in its REVEAL requirements engineering method.[17] More specialized variants of the idea have appeared before, most notably in David Parnas's Four Variable Model.[32]

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

First, the case should be *auditable* so that it can be evaluated by a third-party certifier, independent both of developer and customer. The effort of checking that the case is sound should be much less than the effort of building 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 expertise in software engineering and familiarity with the domain area.

Second, the case should be *complete*. This means that the argument that the critical properties apply should contain no holes to be filled by the certifier. Any assumptions that are not justified should be noted 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 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 under water or that a file synchronizer will work only if applications do not subvert file modification dates. Assumptions made to simplify the case, and that are no more easily substantiated by others, are suspect. Suppose an analysis of a program written in C, for example, contains an assumption that array accesses are within bounds. If this assumption cannot readily be checked, the results of the analysis cannot be trusted.

Third, the case should be *sound*. It should not, for example, claim full correctness of a procedure on the basis of nonexhaustive 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.

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.

Dependability case as product. In theory, one could construct a dependability case *ex post facto*, when the entire development had been completed. In practice, however, this would be near impossible and, in any case, undesirable. Constructing the case is 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 later, 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, every step of the way, to consider their decisions in the light of the system's dependability and to view the evidence that these decisions are sound as a work product that is as integral to the final system as the code itself.

Procurement. A change to a direct approach affects not only developers but also procurers, and the goals set at the start must be realistic in terms both of their achievement and demonstration.

The Federal Aviation Administration specified three seconds of downtime per year for the infamous Advanced Automation System for air-traffic control (which was ultimately canceled after an expenditure of several billion dollars), even though it would have taken 10 years just to obtain the data for substantiating such a requirement.[5] It was later revised to five 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 and to absorb the costs of workarounds.

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. After

an investigation, however, the Navy cited human error. 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'.

With the indirect approach to certification, it has been traditional that procurers give detailed prescriptions for how the software should and should not be developed and which technologies should be used. By contrast, 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.

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 requirements document says can do as much damage as what it fails to say.

In the context of dependability, the approach to requirements is especially important. The standard criteria of course apply: that the developers listen carefully to the stakeholders to understand not merely the functions they say they want but also, more deeply, the purposes they believe these functions will enable them to accomplish; that the requirements be expressed precisely and succinctly; 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 other criteria take on greater significance, however. 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, because (as explained later) the most feasible way to offer high dependability at reasonable cost is to exploit modularity to establish critical properties locally.

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

These two criteria are hardly new, but they are not always followed. Perhaps under pressure from customers to provide an extensive catalog of features, analysts often express requirements as a long list of functions. In a radiotherapy application, for example, the analyst—aware of the risk of delivering incorrect doses or of unauthorized access—might include sections describing the various functions or use cases associated with selecting doses or logging in but might neglect the more important task of describing the most critical properties explicitly—such as that the delivered dose corresponds to the prescribed dose and that access be restricted to certain staff.

Sometimes developers appreciate the value of prioritization but have been shy to make the bold decisions necessary for downgrading (or eliminating) noncritical requirements. Rather than asking "What are the critical properties?" we might instead 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.

Decoupling and simplicity. How much the cost of developing a system will increase if a particular critical property is 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 essence, all of the code becomes critical. But if the property is *localized* to a single component, attention can be focused on that component alone, and the rest of the codebase can be treated as noncritical. Put another way, the cost of making a system dependable should vary not with the size of the whole system but with the extent and complexity of the critical properties.

Decoupling is the key to achieving locality. Two components are decoupled if the behavior of one is unaffected by that of the other. Maximizing decoupling is a guiding principle of software design in general, but it 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, decoupling is addressed by allocating functionality to particular components, which allows key invariants to be localized and the minimization of communication; and by crafting interfaces that do not expose the internals of a service to its clients and do not connect clients to each other 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 design of an e-commerce system, for example, might enforce a rigorous separation between billing and other subsystems; in that way, more complex (and less dependable) code can be written for less critical features (such as product search) without compromising essential financial properties.

Decoupling is an important way of securing simplicity, and its benefits, in system design. As Tony Hoare famously said in his Turing Award lecture (discussing the design of Ada): "[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). They tend to think 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 complicated and critical domain faces difficult problems. The question is not whether complexity can be eliminated but whether it can be tamed so that the

resulting system is 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: "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."[8]

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, 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. For example, 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.[39]

A rigorous process need not be a burdensome one. Because every engineer involved in a project is expected to be familiar with the process, it should be described by a brief and easily understood handbook, tailored if necessary to that project. Rather than interfering with the technical work, the process should eliminate a mass of small decisions, thereby 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.[15]

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. Having a strong "safety culture" can be the major factor in determining how safe the products are, and in fact one study of formal methods found that their success may be due more to the culture surrounding them than to anything more direct.[35] Efforts to build and maintain a strong safety culture can pay dividends. Richard Feynman, in his dissenting report following the *Challenger* inquiry,[10] was effusive in his praise of the constructively adversarial attitude among NASA's software engineers, to which he ascribed the high dependability of their software.

The question is not whether complexity can be eliminated but whether it can be tamed so that the resulting system is 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.

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 mechanistically; truth will have to be assessed by an impartial review not only of the case itself but also 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. Given that the organization that produces the software and the software itself are intertwined, attempts at improving dependability, and efforts to measure it, must take both into account.

Robust foundations. Just as a skyscraper cannot easily be built on sand, 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++ (which provide neither) requires serious justification, and for a critical system the benefits that are obtained in compensation for the loss of safety 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 behavior 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 mistakes and structural flaws.

Complexity in a programming language can compromise dependability because it increases the chance that the program will behave differently from what 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.[18,19]

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 it did not unify primitive types and objects, even though these features had been part of CLU in 1975.[27] When it was realized that these features were essential, it was too late to incorporate them cleanly. It may be curmudgeonly to complain about Java, especially because it brings so many good ideas to mainstream programming—in particular, 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 realize that arguments that are not mechanically checked are likely to be flawed, so their credibility must suffer and confidence in any dependability claims that rely on them must be reduced accordingly.

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 machine includes 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—such as operating system, middleware, and database—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. Thus when high dependability is required, enthusiasm for their use should be tempered by the risks involved.

The (in)significance of code. A dependability argument is a chain with a variety of 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 they might establish the end-to-end dependability requirement. But the overall argument is only as strong as the chain's weakest link.

Many software engineers and researchers are surprised to learn that the correctness of the code is rarely that 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.[29] Problems with requirements and usability dwarf the problems of bugs in code, suggesting that the emphasis on coding practices and tools, both in academia and industry, may be mistaken. Exploiting tools to check arguments at the design and requirements level may be more important; and it is often more feasible, as artifacts at the higher level are much smaller.[21]

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,[11] especially when very high assurance is required. Note too that in the arena of security, code vulnerabilities are responsible for a much higher proportion of failures than in the safety arena.

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, 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. Thus analysis is needed to fill the gap.

Analysis might involve any of a variety of techniques, depending on the kind of property being checked and the level of confidence required. In the last decade, dramatic advances have been made in analyses that establish properties of code fully automatically through the use of theorem proving, static analysis, model checking, and model finding.

How well these techniques will work and how widely they will be adopted remain to be seen. But a number of industrial successes demonstrate that the approaches are at least feasible and, in the right context, effective. 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;[16] and Airbus has used static analysis to show the absence of low-level runtime errors in the A340 and A380 flight-control software.[7]

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

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. For example, inventors of program analyses have long classified their creations 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—a class of tools that are very effective at catching defects, especially in low-quality software, but that usually cannot contribute evidence of dependability because 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.

Even if an analysis establishes a property with complete assurance, the question of whether the property itself is sufficient still remains. For example, eliminating arithmetic overflows and array bounds errors from a program 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.

Among analysis tools, mathematical proof is generally believed to offer the highest level of confidence. An analysis substantiated with a proof can be certified independently by examining the proof in isolation, thereby mitigating the concern that the tool that produced the proof might have been faulty.

Proof is not foolproof, however. When a bug was reported in his own code (part of the Sun Java library), Joshua Bloch found[4] that the binary search algorithm—proved correct many years before (by, amongst others Jon Bentley in his *Communications* column) and upon which a generation of programmers had relied—harbored a subtle flaw. 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 assumption that no integer overflow would occur (which was reasonable when Bentley wrote his column, given that computer memories back then were not large enough to hold such a large array). In practice, however, such assumptions will always pose a risk, as they are often hidden in the very tools we use to reason about systems and we may not be aware of them until they are exposed.

Closing Thoughts

The central message of this article is that it is not rational to believe that a software system is dependable without good reason. Thus any approach that promises to develop dependable software must provide such reason. A clear and explicit articulation is needed of what "dependable" means for the system at hand, and an argument must be made that takes into account not only the correctness of the code but also the behavior of all the other components of the system, including human operators.

Is this approach practical? The cost of constructing a dependability case, after all, may be high. On the other hand, such construction should focus resources, from the very start of the development, 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 indus-

try regards as too costly (such as formal specification and static analysis of code) can actually reduce overall cost.[2]

Similarly, even though the augmentation of testing with more ambitious analysis tools will require greater expertise than is available to many teams today, this avenue does not necessarily increase the cost either. When low levels of confidence suffice, testing may be the most cost-effective way to establish dependability. As the required level of confidence rises, though, testing soon becomes prohibitively expensive, and the use of more sophisticated methods is 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 millstones around their necks. But because a direct approach based on dependability cases gives developers an incentive to use whatever development methods and tools are most economic and effective, the approach therefore rewards innovation.

Acknowledgments
The key ideas in this article come from a National Academies study[22] 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; to our study director Lynnette Millett; to Jon Eisenberg, director of the Academies' Computer Science and Telecommunications Board; and to our sponsors, especially Helen Gill, who was instrumental in making the case for the study. John Rushby and Martyn Thomas deserve recognition for having been long and eloquent advocates of the direct approach. Many of the opinions expressed in this article, however, are my own and have not been approved by the committee or by the Academies.

Thanks too to Butler Lampson, Shari Lawrence Pfleeger, and Derek Rayside, who gave extensive and helpful suggestions on an initial draft of the article; to the anonymous reviewers; and to Hari Balakrishnan, Bill Maisel and Andrew Myers who gave valuable feedback and shared their expertise on particular topics.

A version of this article with a fuller list of references is available at http://sdg.csail.mit.edu/publications.html.

References

1. Aiken, A. and Xie, Y. 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* (Sept. 2005).
2. Amey, P. Correctness by construction: Better can also be cheaper. *CrossTalk*: The Journal of Defense Software Engineering (Mar. 2002); www.praxis-his.com/pdfs/c_by_c_better_cheaper.pdf.
3. Ball, T. and Rajamani, S. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages* (Portland, Oregon, Jan. 16–18), 2002.
4. Bloch, J. Extra, extra—read all about it: Nearly all binary searches and mergesorts are broken; googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.
5. Cone, E. The ugly history of tool development at the FAA. *Baseline Magazine 4*, 9 (Apr. 8, 2002).
6. Cook, R. and O'Connor, M. Thinking about accidents and systems. In *Medication Safety: A Guide to Health Care Facilities*, H.R. Manasse and K.K. Thompson, Eds. American Society of Health-System Pharmacists, Washington, DC, 2005; www.ctlab.org/documents/ASHP_chapter.pdf.
7. Cousot, P. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the Seventh ACM & IEEE International Conference on Embedded Software*. (Salzburg, Austria, Sept. 30–Oct. 3), ACM Press, New York, 2007.
8. Dijkstra, E.W. The tide, not the waves. In *Beyond Calculation: The Next Fifty Years of Computing*, Denning, P. and Metcalfe, R., Eds. Copernicus (Springer-Verlag), 1997.
9. FDA. *Ensuring the safety of marketed medical devices: CDRH's medical device post-market safety program*, 2006.
10. Feynman, R.P. Appendix F: Personal observations on the reliability of the shuttle. In *Report of the Presidential Commission on the Space Shuttle Challenger Accident*, 1986; science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/Appendix-F.txt.
11. Gallaher, M. and Kropp, B. *Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology, 2002.
12. GAO. *Medical Devices: Early Warning of Problems Is Hampered by Severe Under-reporting*. Publication PEMD-87-1, U.S. Government Printing Office, 1986.
13. Geppert, L. Lost radio contact leaves pilots on their own. *IEEE Spectrum 41*, 11 (Nov. 2004); www.spectrum.ieee.org/nov04/4015.
14. German, A. and Mooney, G. Air vehicle software static code analysis—Lessons learnt. In *Proceedings of the Ninth Safety-Critical Systems Symposium*, F. Redmill and T. Anderson, Eds. Springer-Verlag, Bristol, U.K., 2001.
15. Griswold, W. Coping with crosscutting software changes using information transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (Kyoto, Sept. 25–28, 2001).
16. Hall, A. Using formal methods to develop an ATC information system. *IEEE Software 13*, 2 (Mar. 1996).
17. Hammond, J., Rawlings, R., and Hall, A. Will it work? In *Proceedings of the 5th International Symposium on Requirements Engineering* (Toronto, Aug. 27–31, 2001).
18. Hatton, L. and Safer C. *Developing Software in High-Integrity and Safety-Critical Systems*. McGraw-Hill, 1995.
19. Holzmann, G. The power of ten: Rules for developing safety critical code. *IEEE Computer 39*, 6, (June 2006).
20. IAEA. *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama: Report of a Team of Experts*. (Vienna, Austria, May 26–June 1, 2001); www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf.
21. Jackson, D. Dependable software by design. *Scientific American* (June 2006); www.sciam.com/article.cfm?id=dependable-software-by-de&colID=1.
22. Jackson, D., Thomas, M., and Millett, L., Eds. *Software For Dependable Systems: Sufficient Evidence?* National Research Council. National Academies Press, 2007; books.nap.edu/openbook.php?isbn=0309103940.
23. Jackson, M. Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, 2001.
24. Gross, G. E-voting vendor: Programming errors caused dropped votes. *Network World* (Aug. 22, 2008); www.networkworld.com/news/2008/082208-e-voting-vendor-programming-errors-caused.html.
25. Krebs, B. Cyber incident blamed for nuclear power plant shutdown. *Washington Post* (June 5, 2008); www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958_pf.html.
26. Ladkin, P.,Transcriber. *Transcription of Report on the Accident of Airbus A320-211 Aircraft in Warsaw on Sept. 14, 1993*. Main Commission Aircraft Accident Investigation Warsaw; www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html.
27. Liskov, B. A history of CLU. *ACM SIGPLAN Notices 28*, 3 (Mar. 1993).
28. Littlewood, B. and Wright, D. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Transactions on Software Engineering 23*, 11 (Nov. 1997).
29. MacKenzie, D. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001.
30. Maisel, W., Sweeney, M., Stevenson, W., Ellison, K., and Epstein, L. Recalls and safety alerts involving pacemakers and implantable cardioverter-defibrillator generators. *Journal of the American Medical Association 286*, 7 (Aug. 15, 2001).
31. Ministry of Defence. *Defence Standard 00-42: Reliability And Maintainability Assurance Guides, Part 2: Software*, 1997.
32. Parnas, D. and Madey, J. Functional documentation for computer systems. *Science of Computer Programming 25*, 1 (1995).
33. Perrow, C. *Normal Accidents*, Princeton University Press, 1999.
34. Perrow, C. *The Next Catastrophe: Reducing our Vulnerabilities to Natural, Industrial, and Terrorist Disasters*. Princeton University Press, 2004.
35. Pfleeger, S. and Hatton, L. Investigating the influence of formal methods. *Computer 30*, 2 (Feb. 1997).
36. Rockoff, J. Flaws in medical coding can kill: Spread of computers creates new dangers, FDA officials warn. *Baltimore Sun* (June 30, 2008); http://pqasb.pqarchiver.com/baltsun/access/1502776681.html?dids=1502776681:1502776681&FMT=ABS&FMTS=ABS:FT&type=current&date=Jun+30%2C+2008&author=Jonathan+D.+Rockoff&pub=The+Sun&desc=FLAWS+IN+MEDICAL+CODING+CAN+KILL.
37. Salvadori, M. *Why Buildings Stand Up: The Strength of Architecture*. Norton, 1980. See also Levy, M. and Salvadori, M. *Why Buildings Fall Down: How Structures Fail*. Norton, 1992.
38. Slabodkin, G. Navy: Calibration flaw crashed Yorktown LAN. *Government Computing News* (Nov. 9, 1998); www.gcn.com/print/17_30/33914-1.html.
39. Zetter, K. E-voting undermined by sloppiness. *Wired* (December 17, 2003); www.wired.com/politics/security/news/2003/12/61637.

Daniel Jackson (dnj@mit.edu) is a professor of computer science at the Massachusetts Institute of Technology and a principal investigator at MIT's Computer Science and Artificial Intelligence Lab, Cambridge, MA.

© 2009 ACM 0001-0782/09/0400 $5.00