Separation of Concerns for Dependable Software Design

Daniel Jackson and Eunsuk Kang
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139

dnj@mit.edu

ABSTRACT

For 'mixed-criticality' systems that have both critical and non-critical functions, the greatest leverage on dependability may be at the design level. By designing so that each critical requirement has a small trusted base, the cost of the analysis required for a dependability case might be dramatically reduced. An implication of this approach is that conventional object-oriented design may be a liability, because it leads to 'entanglement', and an approach based on separating services may be preferable.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.4 Software/Program Verification; D.2.10 Design.

General Terms

Design, Reliability, Languages, Theory, Verification.

Keywords

Dependability, software design, separation of concerns, object-orientation, formal methods, trusted bases, decoupling, entanglement, mixed-criticality systems.

1. WHY CURRENT APPROACHES FAIL

Today's software developers rely on two techniques to make their software more dependable: *process* and *testing*.

In any large development, a defined software process is essential; without it, standards are hard to enforce, issues fall through cracks, and the organization cannot learn from past mistakes. At minimum, a process might include procedures for version control, bug tracking and regression testing; typically, it also includes standard structures for documents and guidelines for meetings (eg, for requirements gathering, design and code review); and most ambitiously includes collection of detailed statistics and explicit mechanisms for adjusting the process accordingly.

Conference'10, Month 1–2, 2010, City, State, Country.

Testing is used for two very different purposes. On the one hand, it is used to find bugs. Structural tests exploit knowledge of the structure of the software to identify bugs in known categories. A mutation test, for example, might focus on the possibility that the wrong operator was selected for an expression; a regression test is designed to detect the reoccurrence of a particular flaw. For this kind of testing, a successful test is one that fails, and thus identifies a bug.

On the other hand, testing can be used to provide evidence of dependability. In this case, tests focused on particular known bug categories are less useful (since a failure might come from a category that has not been identified). Instead, tests are drawn randomly from the 'operational profile' – the expected profile of use – and statistical inferences are made about the likelihood of failure in the field based on the sampling carried out in the tests. For this kind of testing, a successful test is one that succeeds, since a failing test case might not only require a bug fix (which sets the testing effort back to square one, since the target of the testing is now a new program on which old results no longer obtain), but provides direct evidence of low quality, thus altering the tester's assumptions and raising the bar for demonstrating dependability.

For modest levels of dependability, process and testing have been found to be effective, and they are widely regarded to be necessary components of any serious development. Arguments remain about exactly what form process and testing should take: whether the process should follow an incremental or a more traditional waterfall approach, or whether unit testing or subsystem testing should predominate, for example. But few would argue that process and testing are bad ideas.

For the high levels of dependability that are required in critical applications, however, process and testing – while necessary – do not seem to be sufficient. Despite many years of experience in organizations that adhere to burdensome processes and perform extensive testing, there is little compelling evidence that these efforts ensure the levels of dependability required. Although it seems likely that the adoption of rigorous processes has an indirect impact on

dependability (by encouraging a culture of risk avoidance and attention to detail), evidence of a direct link is missing.

The effectiveness of statistical testing lies at the core of the difficulty, since the most likely way a process might establish dependability would be via testing – in the same way that industrial manufacturing processes achieve quality by measuring samples as they come off the assembly line. Unfortunately, even if the operational profile can be sampled appropriately, and all other statistical assumptions hold, the number of tests that is required to establish confidence is rarely feasible. To claim a failure rate of one in R executions or 'demands' to a confidence of 99% requires roughly $5R$ tests to be executed [5]; similarly, a rate of one failure in R hours requires testing for $5R$ hours. To meet the oft-stated avionics standard of 10^9 failure per hour, for example, would require testing for 5.10^9 hours – almost a million years.

2. A DIRECT APPROACH

In the last few years, a different approach has been advocated, drawing on experience in the field of system safety. The idea, in short, is that instead of relying on indirect evidence of dependability from process or testing, the developer provides *direct* evidence, by presenting an argument that connects the software to the dependability claim.

This argument, which is known as a *dependability case*, takes as premises the code of the software system, and assumptions about the domains with which it interacts (such as hardware peripherals, the physical environment and human operators), and from these premises, establishes more or less formally some particular critical properties.

The credibility of a dependability case, like the credibility of any argument, will rest on a variety of social and technical factors. When appropriate, it may include statistical arguments, although logical arguments are likely to play a greater role. To argue that a component has a failure rate of less than one in 10^9 demands, it may be possible to test the component exhaustively, but if not, testing to achieve sufficient statistical confidence is likely to be too expensive, and formal verification will be a more viable option.

Formal methods will therefore likely play a key role in dependability cases, since there are no other approaches that can provide comparable levels of confidence at reasonable cost. Note, however, that just as rigorous process does not guarantee dependability, the use of formal methods does not either. A formal verification is only useful to the extent that it provides a critical link in the dependability argument. Insisting on the use of particular formalisms, or on the elimination of particular anomalies, is a priori no more likely to be cost effective than the imposition of any other process practice (such DO178B's requirement for MCDC testing). For this reason, the claim that software is 'free from runtime errors' should be taken with a grain of salt. While it is certainly useful to know that a program will not suffer from an arithmetic overflow or exceed the bounds of an array, such knowledge does not provide any direct evidence that the software will not lead to a catastrophic failure. Indeed, the construction of a dependability case may reveal that the risk of runtime errors is not the weakest link in the chain.

3. THE COST OF BUILDING A CASE

A great attraction of statistical testing is that its cost does not depend on the size of the software being checked. Unfortunately, for the analyses that are more likely to be used as the elements of a dependability case, and which rely on examining the text of the software, cost increases at least linearly with size.

This underlines the importance of designing for simplicity. The smaller and simpler the code, the lower the cost of constructing the case for its correctness; the additional upfront cost of a cleaner design will likely prove to be a good investment. But even if Hoare is right, and 'inside every large program there is a small one trying to get out', it may not be possible to find that small program, especially the first time around.

A more realistic aim is to reduce not the size of the entire program, but rather the size of the subprogram that must be considered to argue that a critical property holds. This subprogram, the *trusted base* for the critical property, must include not only the code that directly implements the relevant functionality, but any other code that might potentially undermine the property.

One might think that identifying trusted bases for properties in this way helps only because it allows the rest of the program to be ignored. But an advantage may be gained even if the trusted bases cover the entire program.

Suppose the dependability case must establish k properties of the system, and that each property has a trusted base of size B. The analysis cost is likely to be superlinear; a reasonable guess is that it will be quadratic, to account for the interactions between parts. In that case, the total cost would be kB^2. Now if the trusted bases cover the entire program (but do not overlap), the size of the total codebase is kB. For the same codebase, but without a factoring into trusted bases, the cost of the analysis would be $k(kB)^2$, which is larger by a factor of k^2. Even if the analysis cost were only linear in the size of the trusted base, the cost of analyzing k properties on a codebase of size kB would be k^2B, compared to the cost kB of analyzing k properties, each over a trusted base of size B.

This analysis does not account for the possibility that the effort involved in analyzing different properties might overlap. For example, if the reliability of communications channel is needed for multiple properties, it may be possible to determine just once that the channel is reliable, so that its code is analyzed a single time. In the classical approach to verification (which has yet to be realized in full on a large-scale system), each component is verified with respect to its specification; the high level properties then require only an analysis at the top-level using the specifications of the largest components.

The cost reduction due to this sharing of subanalyses is likely to be a significant factor for systems in which the critical properties that comprise the dependability claims essentially cover the entire functionality. A medical device such as an infusion pump or a pacemaker might fall in this category; it is hard to imagine anything that could go wrong in such a device without compromising its safety.

For most systems, however, different properties will be critical to different degrees, and will therefore call for different levels of confidence and different levels of investment. Moreover, there will be large aspects of the functionality that will not require consideration in a dependability case at all. For example, for an online bookstore, for the functionality that deals with search, advertising and reviews, conventional testing may suffice to establish confidence that the application is deployable. In contrast, the properties that credit card numbers are not leaked, or that customers are billed for the amount indicated, might merit construction of a dependability case.

For these *mixed-criticality* systems, the critical properties are so partial with respect to the overall functionality that there are few opportunities for shared subanalyses; the cost of verifying a component to a specification that is sufficient for the analyses of multiple properties will usually not be justified. Moreover, the high variance in criticality will make it worthwhile to split the system into multiple trusted bases, ideally with smaller bases associated with the more critical properties.

4. DECOUPLING MECHANISMS

The trusted base of a critical property may be larger than desirable for two distinct reasons. One is simply that the code that implements the property is not as localized as it might be. The other is that the relevant code is localized appropriately, but an analysis of additional modules is required in order to determine that their code is not relevant.

(Incidentally, one might think that the notion of trusted base could be defined in execution rather than analysis terms. A module might be included in the trusted base if its failure can undermine the critical property. While such a notion is possible, and was used by Parnas in defining his uses relation [6], it turns out to be extremely tricky to pin down, in large part because 'failure' is not well defined.)

The trusted base idea thus provides a phasing of analyses. In the first phase, a robust but inexpensive *decoupling analysis* – conducted preferably at the design level – determines which modules are in the trusted base and are thus relevant; in the second phase, these modules are analyzed to ensure that the property holds.

In the simplest case, the decoupling analysis can rely on physical isolation, but more often an appeal to a virtual isolation is needed, made possible by a decoupling mechanism. Such mechanisms are available at all levels. The machine and its operating system may provide address space separation, supporting the inference that distinct processes run independently; a middleware platform may provide communication channels with the implicit guarantee that no other interactions are possible; and the programming language may provide namespace access control and strong typing, so that data can be encapsulated.

In all these cases, arguing for decoupling and thus shrinking the trusted base will still require the discharging of some assumptions. For example, in a language with strong typing, establishing lack of interaction between two modules will require a simple argument that their namespaces are disjoint (and if their types overlap, may also require an aliasing/escape analysis). Such an analysis may be non-trivial, but it is generally far cheaper than the subsequent analysis that the relevant modules enforce the desired property, thus justifying the phase separation and the notion of the trusted base.

In contrast, if support for decoupling is lacking, there will be no such cost differential, and a claim for a smaller trusted base will be less useful. For example, in a program written in an unsafe language (such as C), code in any one module can in principle modify data accessed by another (by exceeding the bounds of an array, for example, and modifying arbitrary memory), whether or not there is any overlap in their namespaces.

5. OBJECT-ORIENTED ENTANGLEMENT

A good software design, then, is one that (with the help of available decoupling mechanisms) gives small trusted bases for the critical properties. Finding a good design will obviously require insight, experience and domain expertise. Nevertheless, it is worth asking whether the basic design strategies that are conventionally used are a help or a hindrance.

From the perspective of dependability and trusted bases, it seems that object-orientation may actually be a liability. The standard exhortation to group the common properties

of a domain entity into a single program object may not only fail to aid in decoupling, but may make things worse, by creating an unnecessary *entanglement* of features.

Consider, for example, the design of an online bookstore. A standard domain model would likely include entities such as *Customer*, *Book*, *ShoppingCart*, *Order*, *CreditCard*, etc., and when these entities are realized as classes in the code, the associations between these would likely be represented as fields (typically backed by the tables of a relational database). Worse, these associations, being semantically bidirectional, will often be supported in the code by bidirectional references for easy navigation; for example, the field that obtains a customer's shopping cart will be matched by a field from the shopping cart back to the customer.

As a result, code that should be irrelevant to a critical property cannot be easily factored out. If the property, for example, is that credit card numbers are not inadvertently revealed, the entire *Customer* class will be relevant because one of its fields holds a reference to the customer's card; and the *ShoppingCart* will be relevant because of its back reference to *Customer*.

These observations mirror concerns that have been raised before about object orientation. *Roles* embody the idea that an object has distinct ways of participating in different functions [7]. The *Visitor* pattern [1], and more broadly subject orientation [8] and aspect orientation [3] seek to overcome the 'tyranny of the dominant decomposition' [8], in which all functionality is decomposed across a single object hierarchy that follows the structure of the problem domain.

Rather than fixing the problems of object-orientation, it may be simpler and more effective to structure a system from the outset as a collection of independent services (perhaps corresponding to Jackson's *subproblems* [4]), which are then implemented in a conventional style, and connected together narrow interfaces.

In the bookstore example, searching for books, reviewing, advertising, and billing might each be implemented as a distinct service. To charge a customer's credit card, a coordination component might make a call on the billing service using a customer identifier that is mapped internally by the service to a credit card record. In a monolithic object-oriented implementation, this identifier would be the address of the customer object, and holding it would allow a client access to all features of a customer. Here, in contrast, use of the identifier for billing purposes is controlled by the API of the billing service.

This kind of separation of concerns is not new. But it does not seem to be widely applied, and there is no systematic design method that exploits it to the full.

6. A FOCUS ON DESIGN

In a mixed-criticality system such as an online bookstore, the proportion of the code relevant to a critical property (such as protecting credit cards) is likely to be low – perhaps 5% or less. Designing for a small trusted base might thus decrease the cost of constructing the dependability case by a factor of 20, which is likely far more than can be achieved by advances in the near future in language design, static analysis or verification.

This suggests that as a research field, software engineering might do well to redirect its attention, placing less emphasis on languages and analyses, and more on design and methodology. Research on design is harder to assess, though, and the current taste for empirical evaluation seems to drive research into the areas where potential gains are the most immediate, and thus often the smallest. No wonder that much research is now focused on *ex post facto* analysis, and on the problems of legacy code.

Perhaps it's time to redress the balance, and to focus again on the fundamental problem of design. Analysis can help, but dependability cannot emerge as an accident. It will come for sure only by design.

7. ACKNOWLEDGMENTS

The authors gratefully acknowledge support from the National Science Foundation under grant 0541183 (Deep and Scalable Analysis of Software) and from the Northrop Grumman Cybersecurity Research Consortium.

8. REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[2] Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds. *Software For Dependable Systems: Sufficient Evidence?* The National Academies Press, Washington, DC. 2007.

[3] Gregor Kiczales et al. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[4] Robin Laney, Leonor Barroca, Michael Jackson and Bashar Nuseibeh. Composing Requirements Using Problem Frames. *International Conference on Requirements Engineering*, 2004.

[5] Bev Littlewood and David Wright. Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software. *IEEE Transactions on Software Engineering*, 23:11, 1997.

[6] David Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5:2, 1979.

[7] Trygve Reenskaug, P. Wold and O. A. Lehne. *Working With Objects: The Ooram Software Engineering Method*. Manning/Prentice Hall, 1996.

[8] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. International Conf. on Software Engineering, 1999.