Dependability Co-Design

R. A. Riemenschneider and Steve Dawson
System Development Laboratory
SRI International
333 Ravenswood Av
Menlo Park, CA 94025
{rar,dawson}@sdl.sri.com

Abstract

It is widely agreed that high levels of software system dependability can only be achieved if the system is designed with dependability in mind; security and other dependability properties cannot be "added-on" to an undependable implementation. However, most modern software development methodologies are based on the notion that quality is achieved through incremental improvement. The question then arises: Are these modern evolutionary methodologies incompatible with strict dependability requirements? Our answer is that they need not be, provided dependability concerns are addressed in a semi-independent, but closely coupled, fashion we call *dependability co-design*.

1 Functionality versus Dependability

A software system is *dependable* to the extent that it can justifiably be relied upon to satisfy its requirements. Many system properties contribute to dependability. Security is a representative example: clearly, a system containing serious security flaws – say, one that is easily infected by a "virus" transmitted by email – cannot be depended upon to behave as it should. The vast majority of systems have at least a few strong dependability requirements (e.g., that the privacy of passwords must be protected), and some systems have a wide range of extremely stringent dependability requirements.

Dependabilty properties tend to be properties of the system as a whole, in the sense that a serious flaw in any part of the system can destroy dependability. For that reason, dependability typically cannot be achieved by adding software to an undependable system. Again, security is a representative example. If a single system component allows unauthorized access to information that must be kept private, the system as a whole is insecure. Plugging the information leak generally requires modification of the component's internals, and perhaps a change in the system architecture, rather than adding an additional "plug" component. Even worse, leaks can be an emergent property of the system, in the sense that no individual component compromises information security, but the combination of components does so. Thus, the discovery of dependability problems can indicate that extensive — hence, costly — global changes in the system are needed. As a result, there is considerable incentive to make sure that required levels of dependability are designed-in as a system is developed.

This observation appears to conflict with recent work on software development methodologies, however. Modern methodologies, from the prototyping-based methodologies of the 1980's [1] to today's "eXtreme programming" [2] movement, de-emphasize requirements specification. Rather than trying to specify requirements — a notoriously difficult business with typically unsatisfactory results — one proceeds with programming based on an informal understanding of what the system is supposed to do. Once an executable program is available, the customer can evaluate whether it satisfies his (still unstated) requirements. If not, he can request that specific changes be made, still without having to explain exactly how those changes contribute to requirements satisfaction, much less what the requirements are. The common feature of these methodologies is the emphasis on evolution as the means of achieving requirements satisfaction.

There are two sorts of arguments in favor these modern evolutionary methodologies, one positive and the other negative. The positive argument is that, historically, virtually all systems that are now considered to be high-quality have evolved from earlier versions that were lower quality in response to market demands. The negative argument is that, historically, customers have been unable to fully explicate their requirements. Almost without exception, when development is based on an initial requirements specification and the assumption that any system that satisfies the specified requirements will be acceptable, the customer winds up disappointed by the final product. So the modern development methodologies amount to nothing more than codifications of what experience has shown to work best.

Does adoption of a development methodology that eschews a great deal of "up-front" formal design doom a developer to either extensive redesign and re-implementation to achieve dependability after the desired functionality has been achieved, or living with a software system that functions well but cannot justifiably be depended upon to be safe, secure, reliable, and so on, in every circumstance? We believe that the answer is *no*. The key is to efficiently achieving dependability is to have experts continually assess the developing system, providing advice and catching errors as soon as possible, in a process we call *dependability co-design*.

2 The Dependability Co-Design Approach

The essence of dependability co-design is to have a team of experts constantly assess the dependability implications of design decisions made by the software development team, who focus primarily on achieving the desired system functionality. The dependability experts assessments are both proactive and reactive. They provide explicit guidance to developers allowing them to avoid dependability pitfalls, sometimes in the form of design constraints that must be satisfies, but often in the form of prima facie attractive design alternatives that should be rejected in the next development phase. They also assess recent design decisions from the standpoint of dependability, suggesting changes when necessary. The details of the coordination of development and dependability assessment are somewhat development methodology-specific — although there is a common methodology-independent essence — so fixing a development methodology should make the idea more understandable.

Suppose that our methodology dictates proceeding a top-down fashion — either in a single pass ("the waterfall model") or multiple passes, as in Boehm's spiral model [3] — from requirements specification, through architectural design, functional design, coding, and integration, to deployment and maintenance.[1] How dependability co-design is realized in this context is shown in Figure 1.

At the beginning of the development effort, both functional requirements and dependability requirements are poorly understood. The purpose of requirements specification is to explicate both sets of requirements, at an abstract level. The top-level functional requirements state what the system must do in order to perform its mission. Of course, this statement is not expected to be complete. As development proceeds, further requirements emerge. However, it is expected to be more-or-less correct, and this expectation is not terribly difficult to satisfy in many cases. (Cases in which it cannot be satisfied are poor candidates for a top-down approach, even an incremental top-down approach, to development.) Similarly, the top-level dependability requirements provide a first approximation to what the system must *not* do, given its mission. Functionality and dependability requirements are then elaborated in parallel, each influencing the other.

Initially, the developers produce a system architectural description based on the functional requirements and any advice on potential architectural pitfalls provided by the dependability assessment team. For example, dependability advice might include a recommendation to include a "User ID" argument in all calls triggered by user interaction with the system, to satisfy an authentication requirement. After the architec-

[1] The story is much the same if the process is not actually top-down, but, given the products of the process, it can be "rationally reconstructed" as top-down. This characterizes a great deal of development in the defense and aerospace industries, which is governed by contractual traceability requirements.

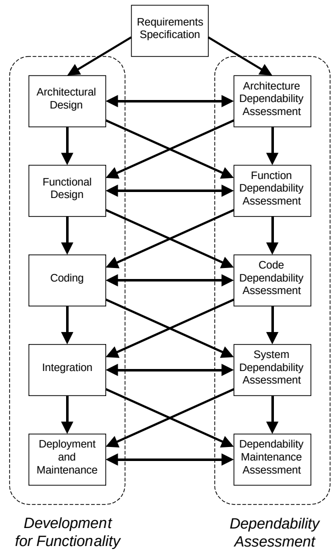

Figure 1: Top-Down Dependability Co-Design

tural description is completed, it is reviewed by the dependability assessment team, and feedback is provided to the development team before the system design is further elaborated. Feedback might include advice that an argument in a call across machine boundaries be encrypted, to satisfy a confidentiality requirement. As a result, architectural design decisions that would have a negative impact on dependability can be avoided or immediately corrected, reducing the likelihood that the architecture will make dependability requirements unsatisfiable. Next, a functional description that elaborates the architectural description is developed, based on guidance provided by the dependability assessment team. That guidance is in turn based upon dependability analysis of possible elaborations of the architecture. Once a tentative functional description is completed, the dependability impacts of the design decisions it is based upon are reviewed. This combination of proactive and reactive advice to the development team continues until the system is delivered to the customer (and even after, during maintenance).

The generalization of this style of interaction between the developers and the dependability assessors to other development paradigms is straightforward. A stage in the development consists of making design decisions that are embodied in the further elaboration of some system artifact (e.g., system source code).[2] That elaboration is guided, in part, by advice from the dependability assessment team, who have analyzed all pre-existing artifacts and knows what mistakes are most likely to be introduced in the next stage. After the elaboration has been tentatively completed, the impact on de-

[2] A division into stages is not really essential, since the emphasis is on more or less continual interaction between the two teams, but it simplifies the explanation.

pendability other design decisions is analyzed, modifications are suggested, and advice regarding the next stage is produced.

3 Technology Insertion

One of the main advantages of this approach to ensuring dependability is that it does not require adoption of a new development methodology. Moreover, it can be adopted incrementally. The size of the dependability assessment team and the frequency of interaction between that team and the development team can be adjusted according to the stringency of the dependability requirements for the system. Initially, an organization might employ a single dependability "guru" to assess the dependability of completed designs or prototypes, an entirely reactive approach that would have comparatively little impact on the development effort. This limited experiment could establish the value of independent dependability reviews, and point out concrete instances of errors that could have been avoided by more frequent, proactive analyses.

Dependability assessment requires different skills, different tools, and a different mind set than development. One of the lessons that the modern methodologists teach is that programmers are most productive when they use a generate-and-test methodology: write some code, then test it and debug it until it seems to work. Dependability requires that certain events must never occur, under any circumstances. Testing is fundamentally ill-suited to establishing dependability in complex systems. More formal methods — such as model checking [5] and theorem proving [4] — are needed, together with plenty of experience and common sense.

Instead of attempting to make every developer an expert in dependability assessment, security co-design suggests that a dependability oracle be provided for developers to consult. In so doing, dependability co-design build on the tradition of having an independent test team evaluate the finished program. The principal differences are that this independent evaluation is extended to every stage in the development lifecycle, and that an attempt is made to avoid introducing errors, as well as discovering errors that have already been made.

References

[1] W. W. Agresti, *New Paradigms for Software Development*, IEEE Computer Society Press, Order No. 707, 1986.

[2] K. Beck, *eXtreme Programming eXplained: Embrace Change*, Addison Wesley, 1999.

[3] B. Boehm, "A spiral model of software development and enhancement", *IEEE Computer*, vol. 21, no. 5, May 1988, pp. 61–72.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Shrivas, "A tutorial introduction to PVS", *Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, FL, April, 1995.

[5] G. J. Holtzman, "The model checker Spin", *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997, pp. 279–295.