

REUSE, RELIABILITY, AND SAFETY

Michael Tortorella
Industrial and Systems Engineering
Rutgers University
Piscataway, NJ 08854
mtortore@rci.rutgers.edu

1 INTRODUCTION

1.1 *Rationale*

Life in the modern industrial state would simply not be possible without the complex systems of hardware and software that monitor, control, and execute essential infrastructural processes. All too often, unanticipated undesirable outcomes surprise users and builders. Examples range from the trivial (screen freezes on a DVD replay) to the catastrophic (Chernobyl).

Some stakeholders would prefer more predictable and stable systems. Software reuse has been advanced as a powerful tool that, in the best of circumstances, transfers known good qualities of a proven software product to a new product still in development. Presumably this would also apply to reliability and safety properties. A careful examination of the interactions between reuse, reliability, and safety is required to build confidence in this approach.

1.2 *Scope*

In this paper, we will offer some ideas for such an analysis from the (perhaps idiosyncratic) point of view of a mathematician and reliability engineer whose specific exposure to software reuse is limited to ongoing conversations with leading players in the field. The intent is to clarify what is known so far, harmonize the discussion of software reuse with key concepts of reliability engineering, and prepare a rigorous intellectual foundation for further study. We examine the similarities and differences pertaining to reuse of hardware and software. The key concept is specification of functionality and its linkage to desired system behavior.

The paper is by no means comprehensive or complete. References are omitted, arguments are sketches and sometimes not in proper order, and data are not present. Our hope is that the thoughts expressed may serve as a springboard for more rigorous analysis.

1.3 *Background*

Software reuse has been widely studied and frequently applied. The basic contours of the technique are now well established and those wishing to employ the technique will find no lack of publications, practitioners, and other resources to draw on to help with institution of a software reuse program. Therefore, we may consider that software reuse has reached a high level of maturity and acceptance, and this fact alone provides evidence that software reuse provides numerous tangible and intangible benefits.

Software reliability, while easily defined, is less easily achieved. Parts of the mathematical theory of reliability have been applied to software products with occasional (in this author's opinion, mostly accidental) good results. Some would consider this a case of the classic *post hoc, ergo propter hoc* fallacy because the phenomenological connection between the mathematical reliability models employed and the activities in the software development process they purport to describe is often tenuous at best. It is perhaps a case of the mathematics running out ahead of the engineering, because the standard reliability engineering approach of understanding failure modes, failure mechanisms, preventive actions, and risk analysis is less prominent in the software reliability engineering community than elsewhere, while numerous mathematical

methods for software reliability estimation and prediction fill the journals. Mature understanding of the basic phenomena at play in any engineering systems domain quickly leads to the development of a design for reliability process and discipline, based on such understanding, for that domain. In many cases, it is not possible even for hardware designers to claim that they use systematic design for reliability techniques in their work, but in the software development community the phrase "design for reliability" is still a by and large foreign one. We find ourselves in the undesirable situation where there are many mathematical models for measuring (a restricted subset of) software reliability but far less use of systematic, disciplined procedures for producing reliable software in the first place.

The intensity of the discussion changes when safety is introduced. However, we shall see that the difference between reliability and safety is largely a difference of degree and not a difference of kind. Tragic though the many instances of software failures leading to injury and death may be, and they are, from a technical point of view the solutions that need to be considered are similar for both cases. Software safety has come to be seen as a separate class of problem; many books and papers advance this point of view. It is our challenge to integrate the patterns of thought in these two areas and demonstrate that they are more alike than different. In particular, when reuse is considered in this arena, the questions that need to be addressed regarding reliability and safety are mostly the same.

1.4 Synopsis

In order to prevent an undesirable outcome, the possibility of that outcome must be anticipated and the steps leading to it fully understood so that effective blocking actions may be taken. At the most abstract level, use of historical information about behavior of past systems to anticipate behavior of a new system is a form of reuse. Indeed, historical information is a key foundational element of a disciplined design for reliability scheme because avoidance of past mistakes is a major part of design for reliability.

A software safety failure is a software failure that causes injury or death. Therefore, software safety failures are a subset of all software failures. A software failure is an instance of the software's operation in a manner inconsistent with requirements. Leaving only implicit a requirement that no use of the software shall cause injury or death may lead to overlooking that requirement and inattention to design for reliability and, in this special case, safety. A case can be made that every software object must explicitly include a requirement that no injuries or deaths result from use of the software in any fashion (perhaps this should be waived for weapons systems). This is an instance of reuse at the requirements level.

If the software is an agent or controller for some other system, then the downstream system can be blocked from performing any action injurious to life or health regardless what the controller software may direct. For example, the Therac machine could have been blocked by self-contained means (*i. e.*, independent of the software controller) from ever emitting any radiation in excess of a safe level. If this had been implemented (and was operating correctly), the deaths due to the software controller's failure may have been avoided. This is an example of a reusable design for safety principle.

Software safety failures are different in degree, but not in kind, from ordinary software failures. Design for safety is a subset of design for reliability in software, directed specifically at the requirement discussed above that no injuries or deaths result from use of the software in any fashion. Reuse of proven software artifacts will be a powerful design for reliability tool for software because reused artifacts that have been certified as to context, functionality, and requirements provide a library of resources known to behave correctly under defined conditions.

2 FOUNDATIONS

2.1 Definitions

2.1.1 Software Reuse

Everyone at this conference knows what software reuse is. Indeed, it would probably not be difficult to get even a moderately well educated layperson to deliver a cogent description. For purposes of analysis, we need to understand software reuse in precise terms. For this, the definition advanced by Frakes, "the use of existing software knowledge or artifacts to build new software," is our starting point. In addition, we further consider "software reuse" to include not only the basic concept of taking a lump of existing software knowledge or artifacts and using it again in another program but also the process by which this action is facilitated.

2.1.2 Software Reliability

Reliable software operates correctly according to its specifications throughout its useful life. Note that this concept precedes the usual definition offered which refers to the probability of this occurrence. Our concern here is with irregularities that may occur in the software and its use that precede use of probability models to describe how often irregularities occur.

An instance of the software's not operating correctly according to specifications is a *failure*. Note in this formulation the primacy of specifications. For any approach to software reliability to be sensible, requirements must be complete and cover all possible anticipated uses of the software. Plainly, this is not an easy thing; my colleagues on this panel make the same point in other ways. However, if we do not know how we want the software to operate under certain given conditions, then we have no right to complain when it does something untoward. Should the software be operated in a way inconsistent with specifications, the notion of failure is vacuous and we do not consider as a failure (or a success) anything that happens when the software is so operated. When specifications are incomplete or do not cover all possible anticipated uses of the software, the software suffers from inadequate quality. The behavior of the product over time for as long as the owner wishes to continue using it is the essential characteristic of reliability. The distinction here is that should the software (or any product, for that matter) cease at some time to perform a function that it has successfully performed (or could have performed, although this might be unknowable) in the past, then we say a reliability problem or failure has occurred. If the software, or other product, does not perform some desired action because it was never capable of performing such an action (by omission or commission), then we say that a quality problem exists.

We deliberately avoid introducing notions of probability into the reliability discussion primarily to focus attention on the primary essential elements of software reliability, namely, the understanding of failure modes and failure mechanisms. Knowledge of these must precede construction of any probabilistic model. It may be considered that the source of randomness in such probability models is the unpredictable nature of the sequence of operations to which the user will subject the software, but exploration of this topic is postponed until after clear understanding of failure modes and failure mechanisms and their relationship to software reuse is achieved.

2.1.3 Software Safety

Certain software applications may behave in ways that lead to injury or loss of life. The software is called *safe* if such behavior cannot occur. If the use of the software in the fashion that led to the undesirable behavior was anticipated in the requirements, then this behavior is a failure. Otherwise, to call it a failure is to invoke an implicit requirement that says that no behavior allowing injury or loss of life is permitted.

Phrased this way, safety is an absolute: a lump of software is either safe or it is not. We may introduce the distinct concept of *provably safe*, that is, to the degree that we are able to assess

the validity of the statement that unsafe behavior cannot occur. Should we adopt this approach, the challenge for software engineering is to fully anticipate all possible scenarios in which such failures may occur and to develop the software in such a way as to force only correct operation in those scenarios.

3 ANALYSIS

3.1 Design for Safety

As noted in Section 2.1.3, failures of a software application that lead to injury or loss of life are considered safety problems. Safety problems are merely (not the best choice of words!) a subset of all failures, and as such, differ only in degree, and not in kind, from failures overall. We may choose to invest extra effort in preventing safety failures because of their more severe consequences, but the actions we take in doing so are not markedly different from standard design for reliability practices, with the addition that we now must attempt to anticipate the ways in which the software may be misused (caused to operate on inputs that are not legal according to the requirements). However, the same pattern of anticipating and preventing possible failures obtains. Therefore, design for safety is a subspecies of design for reliability. The problems encountered are no longer in the main technical problems; the design for reliability paradigm is no longer novel. By and large, we know what to do. When we do not do it, it is largely a management failure or a process inadequacy.

Carry this a step further. It would be a cynical business model indeed that permitted explicit tradeoff of gains and losses if safety problems, having been identified (or not, through deliberate blindness), were permitted to occur at some "acceptably low rate of occurrence." Since no one in a litigious society would dream of defining "acceptably low rate of occurrence," it becomes impossible to address the problem technically; whatever you do is not enough: there is no stopping rule. Therefore, design for safety becomes a moral issue because no victim of a safety problem should be asked to be the bearer of the direct cost of the failure to adequately address safety.

3.2 Requirements

Failures, including safety failures, are ontological entities only insofar as the possibility of their occurrence is at least implicitly contained in requirements. However, because of the increased severity of safety-related problems, developers may additionally attempt to anticipate misuses of the software that may cause injuries or loss of life. Strictly speaking, this is outside the scope of design for reliability as understood here, because we do not define "failure" except as a response to an input condition that is legal according to requirements. However, the more serious consequences of safety failures drive us to consider also uses (misuses?) of the software that are not covered in the requirements.

3.3 Software Reuse and Design for Reliability and Safety

The naïve expectation is that reuse of lumps of software that are proven to be failure-free will lead to a more reliable (new) product. This assertion has not yet received full scrutiny, nor has it been tested in experiments that gather reliability data from comparable lumps of software that differ only in whether they reuse other proven failure-free software lumps. Work along these lines must be carried out in order to establish the degree to which this assertion may be relied upon, and to sharpen our understanding of how the reuse process may be arranged for maximum effectiveness. Because safety failures are failures, the degree to which this can be done for reliability should be replicable for safety.

4 CONCLUSIONS

Distilling the above discussion, we may assert certain principles:

1. When it is possible to anticipate and prevent safety problems, this action is accomplished in the same way any other failures are anticipated and prevented: through focused design for reliability actions.
2. To address the explicit or implicit requirement that safety problems must not be allowed to occur, it is necessary to also anticipate how owners might misuse the software.
3. The design for reliability process (sequence of steps) is well understood in the reliability engineering community; its application to software products needs to be further codified and encouraged.
4. Work needed to address the degree to which the reliability of a software artifact transfers to a new product reusing that artifact needs to go forward.