Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

341

Sergio Bittanti (Ed.)

Software Reliability Modelling and Identification

Springer-Verlag

PREFACE

Where are we in the development of applicable methods for the assessment of Software Reliability ? In the attempt to reply to this question, an intensive course on Software Reliability Modelling and Identification was held at Villa Olmo, Como (Italy) from September 2 to 4, 1987, under the aegis of the Centro di Cultura Scientifica A.Volta (Como). The lecturers were Paolo Bolzern, Carlo Ghezzi, Bev Littlewood, John Musa and Riccardo Scattolini, besides the editor of this volume.

The course, which was attended by field engineers, software managers and university researchers, was organized by the Politecnico di Milano, Dipartimento di Elettronica, Centro Ingegneria dei Sistemi per l'Elaborazione delle Informazioni of the Italian Research Council (C.N.R.) and Centro di Teoria dei Sistemi of the C.N.R..

This volume contains five tutorial papers summarizing the content of the various lectures. The purpose is to present the basic models used to forecast the reliability growth during the software testing process, and discuss the practical applicability of models in the management of the software development. Particular attention is paid to the main techniques for the model identification from data (parameter estimation, selection of complexity, validation, etc.). The general approach is to present good theory for the user, in simple and introductory terms.

The monograph organization is as follows. In Chapter 1, reliability is placed in the context of other relevant software qualities. Furthermore, the techniques which have been developed so far to produce a-priori reliable software (constructive approach) are introduced. In spite of the increasing interest for the constructive approach, debugging still takes a significant percentage of time in the life-cycle of a software product. The basic reliabilty concepts (hazard rate, mean time to failure, etc.) as well as the ideas behind the use of mathematical models for software reliability growth are the subject of Chapter 2. Then,two important models are extensively discussed in Chapter 3.

A flexible modelling approach is proposed in Chapter 4. Precisely, a model which can be used to describe a variety of different reliability trends is introduced; flexibility is achieved by allowing a variable fault exposure coefficient, and resorting to simple decision rules to simplify the model when advisable. Finally, Chapter 5 deals with the provision of tools to assist the user for the selection of an appropriate model amongst the main ones proposed in the literature. These tools are based on the analysis of the predictive capability of the various models.

The editor expresses his sincere acknowledgement to the fellow authors for their most valuable contributions, as well as their care and patience in the preparation of manuscripts. He is grateful to Professor Carlo Ghezzi, who originally brought to his attention the problem of software reliability modelling.

The support of the Ministry of Education (M.P.I.) and the C.N.R. stategic project Matematica Applicata is acknowledged.

Milano, October 7 1988

Sergio Bittanti

TABLE OF CONTENTS

CHAPTER 1

ON THE ROLE OF SOFTWARE RELIABILITY IN SOFTWARE ENGINEERING

C. Ghezzi, A. Morzenti, M. Pezzè
Dipartimento di Elettronica
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy

ABSTRACT

We place reliability in the context of other relevant software qualities and try to define it rigorously. Then we discuss two complementary approaches to reliability: the constructive approach, which tries to produce a-priori reliable software, and the analytic approach, which tries to measure reliability by inspecting software a-posteriori, after its development.

The paper reviews two relevant technologies that may provide a constructive contribution to improving software reliability: formal specifications and programming languages. Although our emphasis is on constructive approaches, we briefly review the principles and techniques of software validation, that can be used to check software reliability after development.

1. INTRODUCTION

There is much evidence that software is an extremely complex artifact. We will substantiate this claim here by quoting two eminent software scientists: F. P. Brooks and D.L. Parnas. Fred Brooks says that "software entities are more complex for their size than perhaps any other human construct" [6]. Dave Parnas [20]

observes that the very nature of complexity stems from the fact that software systems cannot be analyzed by well-understood mathematical formalisms, such as the mathematics of continuous functions, as in most traditional engineering fields. Software systems are discrete state systems with an enormous number of states and almost no regularity in their state structure. "The mathematical functions that describe the behavior of these systems are not continuous functions, and traditional engineering mathematics does not help in their verification". Both Brooks and Parnas are rather skeptic on the possibilities of real advances in the field. According to Brooks, "there is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity". Brooks sees no startling breakthroughs, as he believes that this is "inconsistent with the nature of software". Parnas notices that "the state of the art in software engineering is significantly behind that in other areas of engineering. When most engineering products have been completed, tested, and sold, it is reasonable to expect that the product design is correct and that it will work reliably. With software products, it is usual to find that the software has major "bugs" and does not work reliably for some users. These problems may persist for several versions and sometimes worsen as the software is "improved". While most products come with an express or implied warranty, software products often carry a specific disclaimer of warranty. The lay public, familiar with only a few incidents of software failure, may regard them as exceptions caused by inept programmers. Those of us who are software professionals know better: the most competent programmers in the world cannot avoid such problems". He goes on showing that it is not just a matter of maturity of the methods and tools used by software professionals that make software production so critical: it is the conceptual complexity of software that makes software intrinsically unreliable.

These quotations from the literature give convincing arguments on the intrinsic difficulties of the software development process and the resons why software products are less reliable than other engineering products. On the other hand, we must be aware that many

real-world critical applications often rely upon software, and their required quality level farther exceeds what can be assured by current practices. In addition, reliability is only one of the desired qualities of software products; other quality factors include the ergonomics of human interaction with the application, or the ease of evolution of the application to satisfy changing requirements, both at the user's level and at the architectural level. We will discuss more generally the factors affecting the quality of a sofware product in section 2, in order to place reliability in the proper context. Reliability and other related concepts --such as correctness and safety-- will be defined in section 3.

All of the papers in this volume deal with reliability. However, this paper is complementary to the others, which deal with the models that can be used to describe, evaluate, and predict reliability. Here we concentrate on constructing reliable software. Although we agree with Fred Brooks that no existing technology can cause orders-of-magnitude improvements in software quality, we do believe that some existing technologies can make the quality of the resulting software products much better than the quality resulting from conventional practices. Thus, section 3 will review two such technologies: the specification technology and the programming language technology. They will be discussed with emphasis on the reliability of the resulting software.

Although the emphasis of our presentation is on constructive methods --i.e. methods that help us producing high-quality software-- we will complete our presentation in section 4 with a quick overview of the verification and validation technology, which complements constructive methods very naturally.

This paper is tutorial in nature. However, it is not our goal to cover all constructive methods and tools that can help us produce reliable software; rather, we concentrate on the ones we trust more. Similarly, we will not review all approaches to verification and validation, but rather we will stress how they should complement constructive methods to improve the overall quality of software products.

2. SOFTWARE QUALITIES: SYNTHESIS VS. ANALYSIS

Figure 1 describes the relevant software qualities; it is inspired by the taxonomies proposed by [7] and [4]. Software qualities listed in fig. 1 are accompanied by a brief informal description, which leaves much space open to different interpretations. However, as our understanding of software becomes deeper and deeper, we may expect to be able to define them in a precise, formal way. The purpose of defining the exact meaning of the relevant software qualities is to use them as a basis of a rational and rigorous process of software production.

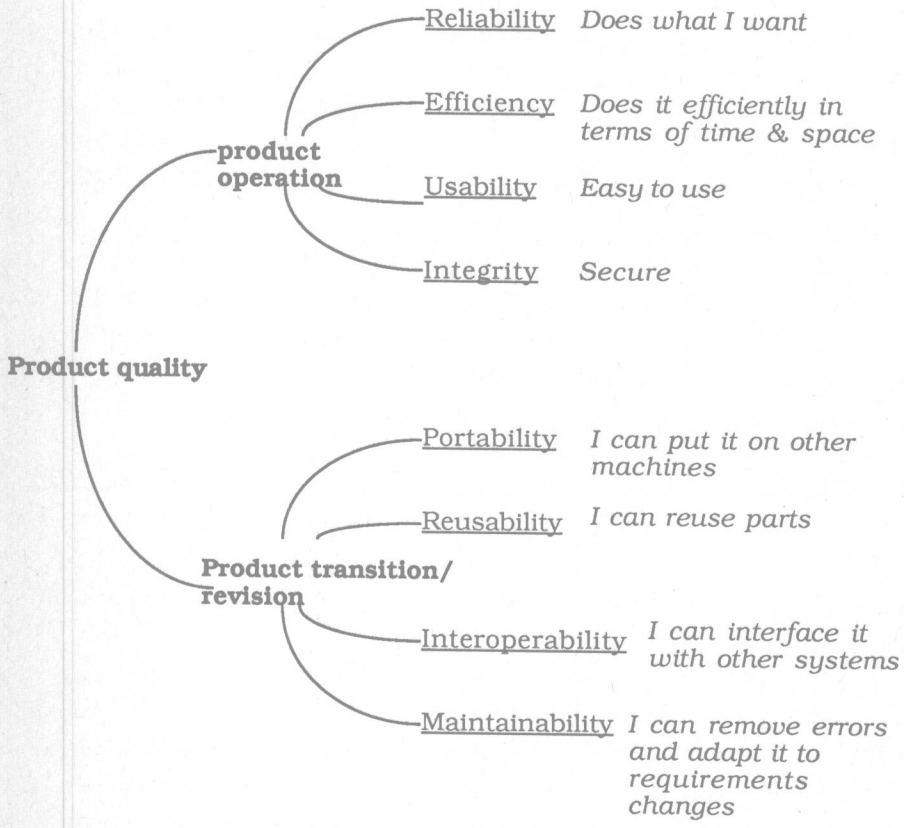

fig. 1: Taxonomy of software qualities

In an ideal design process, one should be able to:
1. **define** the desired level of software qualities in the requirements phase;
2. **design** software using a quality-directed methodology;
3. **measure** the actual software qualities in the resulting product.
In step 1 one should scan a checklist of software qualities, such as the one shown in fig.1, and weight each attribute according to the requirements for the current project. In an ideal world, the weighting scale would be formally defined since software qualities would be formally specified. Step 2 assumes that, once the exact nature of the relevant software qualities is fully understood, suitable design methodologies are available for adoption, which can assure that the required quality levels are indeed achieved by the product being designed. Although this step cannot be done in a fully mechanical way, even in an ideal world, effective heuristics would be avalilable to support design, based on previous experience, as it happens in most traditional and well-established engineering fields. Since step 2 cannot be accomplished entirely automatically, it is subject to human judgement and fallibility. Consequently, a final validation of the product against the goals initially stated in step 1 is still needed; this is done in step 3. Ideally, the measure can be done rigorously and quantitatively, since software qualities are defined formally.
This rational ideal process represents a model that can hardly be approximated by practice in the current state-of-the-art. One basic reason is that most software qualities are defined in a fuzzy way and need to be better understood before they can be defined rigorously. As a consequence, the catalogue of quality-directed design methods cannot be available and exact measuring procedures cannot be devised to support the final validation.
This motivates the need for research aiming at a better understanding of software qualities, in order to be able to formally define them. Once this is done, one will be able to use them as a sound basis for the two complementary approaches described by the above steps 2 and 3. Step 2 embodies what we call the **constructive approach**: the required software qualities are achieved through a quality-directed design process. Step 3 embodies the **analytic**

approach: software qualities are measured a-posteriori in the validation phase.

The two approaches are clearly complementary. In fact, as we have seen, the methodology driving the design process can only be a general guideline; it cannot be seen as an instruction kit. Thus, much is left to human judgement, or even personal taste. Since humans are fallible in applying the methodology, a validation phase is needed to assess the product after development in order to check whether it matches the prescribed quality levels.

At present, several methodologies have been proposed in the literature to derive software from its requirements in a systematic fashion and several tools are offered to support software development. Unfortunately, there is little consensus on which methodology offers the best solution to our design problems and there is no way to evaluate the benefits and trade-offs of different kinds of tools in a precise and quantitative fashion. Here is another area where the constructive and the analytic approaches can complement one another. The availability of analytic tools to evaluate software after development allows the effectiveness of different methodologies to be assessed precisely in terms of the quality of the resulting programs. Similarly, the effectiveness of different tools may be evaluated by measuring the variations in the quality of the resulting products.

The analytic approach is based on the following concepts:

- **metrics**: they are a set of precisely definable software attributes;
- **measures**: they are objective and mechanical ways to determine the values of selected metrics;
- **models**: they are mathematical laws relating entities of a metric.

Notice that analytic approaches are not used only to assess software qualities; the models they provide may be used for prediction. Examples of predictive capabilities can be seen in existing models for cost estimation [5] and reliability models. For example, cost models predict the number of man-months needed to develop some software, based on the estimated number of delivered source instructions, the type of application, and several other factors

(such as the ability of developers, the type of technology used for development, etc.). The case of reliability models will not be treated in this paper: it will be the subject of all remaining contributions in this volume.

3. RELIABILITY AND RELATED CONCEPTS

In the previous section we mentioned one important aspect of software quality: how the product conforms to the prescribed functional specification (i.e., relationship between inputs and expected outputs). This concept will be investigated further in this section; in particular, we will define the concepts of correctness, reliability, security, and safety.

Correctness is a well-understood software quality, that has been defined in various formal settings. In particular, we use here the so-called axiomatic definition, which is based on logic.

A program *P* is specified as a function f_P from an input domain *In* to an output domain *Out* :

f_P : *In* -> *Out*

where input data belonging to *In* and *Out* are described by predicates *PIn* , *POut* , respectively. P is **correct with respect to its specification** if, for every input satisfying *PIn* , *P* terminates in a state satisfying *POut* .

For example, a sorting program operating on a nonempty input array *a* of length *n* can be specified as follows:

```
PIn  :   n >= 0
POut  : a[i] <= a[i+1] forall i with 1 <=i< n  and
        is_permutation(a, a')
```

where *a'* denotes the value of *a* before execution of sort and *is_permutation* is a predicate specified elsewhere whose intended meaning is obvious.

There is an important concept hidden in the above definition: The specification is assumed "correct" (adequate) by definition, and correctness is defined as relative to the specification. Also, the definition of correctness given here should be read as "correctness

with respect to the prescribed functional behavior (input/output relationship)". Although usually this is the intended meaning of the term in the literature, one could use the term more generally as conformance of a given program to some formally specified quality, such as response time, or storage requirements, or even usability, if one can provide such formal specification.

Figure 2 outlines the software production process in a sketchy form. It distinguishes between two main activities: specification and implementation. Also, it shows that the nature of checking whether requirements satisfy the intended user requirements is intrinsically non-mathematical. We can help the process of assessing requirements by providing appropriate notations that make requirements more understandable; we can even try to make requirements "executable" or to use them to derive a prototype.

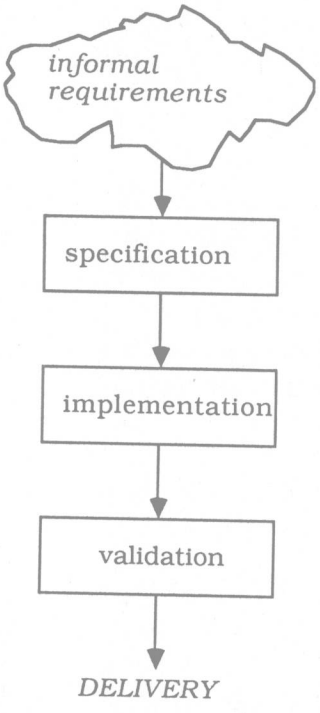

fig. 2: A rough view of the production process

However, the customer can only assess the requirements by comparing the intuitive ideas in his or her head with the more or less formal description provided by the software specialist and by examination of the prototype's behavior, if there is one. Instead, the subsequent implementation can be checked for correctness with respect to the specified requirements. This process will be examined in more detail in the next section. However, we wish to stress again that the correctness as defined here does not imply that the observed functional behavior of a correct software will correspond to the user's expectations. Obviously, this is the desired overall goal, but it is useful to split it into the two subgoals:

a) producing good requirements specification; and
b) producing correct software with respect to its specification.

According to this distinction, one can invest in the direction of goal a) by providing methods and tools that help capturing, specifying, and assessing requirements. This is a currently active research area, which emphasises also the use of graphical interfaces to facilitate human interaction and the use of rapid prototyping techniques or diagram animation to inspect the evolution of the specified system. Goal b) is then facilitated by the (potentially) formal nature of the process. In fact, both the specification and the implementation can be viewed as formal notations: transformation techniques may be used to transform one into the other, and formal proof systems can be devised to demonstrate their equivalence.

The concept of correctness defined here may be too strong to be achieved in practice. Absolute adherence of a program to its specification may be too difficult to achieve and often perhaps also not mandatory. What really matters in many practical cases is some sort of probabilistic relaxation of the concept of correctness, in order to express the fact that the user of the application can safely *rely upon* its usage. Here is where reliability comes into our framework. Formally stated, **reliability** is the probability that software will behave correctly --as specified by the requirements-- for a given time interval. Again, since correctness is usually intended with respect to functional behavior, the same applies to reliability, although in principle a probabilistic

measure might be applied also to other characteristics of software quality.

Formally, the adjective "reliable" attached to software has no precise meaning by itself: a figure must be attached to it to denote the required (or measured) probability. In practice, however, the term is used to denote loose concepts, such as "likely absence of failures", or "long execution histories without failures", etc., where **failure** denotes a visible deviation of the program's behavior from what is specified.

Often, people use related concepts like safety or security. The term **safety** is used to denote the absence of undesirable behavior causing system hazards. Usually, it deals with requirements other than those dealing with the primary mission of a system and requires ensuring that software will execute without resulting in unacceptable risk. [19] is an excellent tutorial on software safety. From a formal viewpoint, safety properties are just any set of properties that one can specify in the requirements; having done that, one can then try to ensure correctness of the implementation with respect to this specification. Of course, the properties to specify when dealing with safety do not simply deal with correct functional behavior in terms of input/output relationship. Often, they describe what should never happen while the software is executing (this is also called a *negative requirement*); in some cases, the requirement is stated as an *invariant property* of the system. For example, a safety requirement for an aircraft is that its height should alway be greater than or equal to zero. Correctness with respect to an invariant property requires a modification of the definition given above when dealing with correct functional behavior specified axiomatically, since every state entered by the system must satisfy the invariant predicate.

Security is highly related to safety: it has been used mainly to denote unauthorized access to classified information. [18] shows that security can be formally specified and implementations can be proven correct with respect to the specification. Both safety and security are such that a probabilistic measure of their correctness (i.e., reliability) does not appear to be interesting: in fact, the

system's behavior is totally unacceptable as soon as it deviates from its specification, no matter how often this happens.

3.1 Producing reliable software

We have defined the concept of *correctness with respect to a specification* and we have introduced reliability as a relaxation of correctness that can be defined in probabilistic terms. Lastly, we have analyzed the related concepts of safety and security and we have seen that they can be viewed as special cases of the concept of correctness, where the specified property is not the input/output functional behavior, but the statement that some hazardous or unsecure state is never entered during execution.
In this section we will return to the informal and loose usage of the term reliable and we discuss some existing technologies which, in our opinion, can improve software reliability significantly. We discuss formal specifications in section 3.1.1 and programming languages in section 3.1.2.
Specifications are intrinsically related to our subject matter, since correctness was defined as a property that is relative to its specification. In this paper, it will be impossible to review all relevant approaches to specifications, from informal to formal, from those based on textual notations to those highly graphical and interactive. This is presently an active research area and much attention is devoted to the definition of tools supporting specification, helping the designer to progress from informal to progressively more formal specifications [12] [8] [14]. Our viewpoint is that the goal of specification should be to derive a rigorous description of the requested behavior: the more critical or less understood the system under study, the more formal the description should be.
The benefits of formal specification with respect to software reliability can be summarized as follows:

(a) they support a rigorous, unambiguous description of the expected functional behavior;

(b) they support a better understanding of the problem at hand, as a side-effect of the effort to produce a formal description;