Software Reliability

Conference Paper · January 2005

DOI: 10.1007/978-3-540-68947-8_10 · Source: DBLP

CITATIONS
17

READS
5,509

5 authors, including:

Irene Eusgeld
University of Duisburg-Essen
19 PUBLICATIONS **665** CITATIONS
SEE PROFILE

Matthias Rohr
BTC Business Technology Consulting AG, Germany
32 PUBLICATIONS **466** CITATIONS
SEE PROFILE

Felix Salfner
SAP Research
37 PUBLICATIONS **1,183** CITATIONS
SEE PROFILE

Ute Schiffel
Reykjavik University
20 PUBLICATIONS **288** CITATIONS
SEE PROFILE

10 Software Reliability

Irene Eusgeld[1], Falk Fraikin[2], Matthias Rohr[3], Felix Salfner[4], and Ute Wappler[5]

[1] Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
[2] Darmstadt University of Technology, Germany
[3] University of Oldenburg, Germany
[4] Humboldt University Berlin, Germany
[5] Dresden University of Technology, Germany

Many concepts of software reliability engineering can be adapted from the older and successful techniques of hardware reliability. However, this must be done with care, since there are some fundamental differences in the nature of hardware and software and its failure processes. This chapter gives an introduction into software reliability metrics.

10.1 Introduction

Software reliability is often defined as "the probability of failure-free operation of a computer program for a specified time in a specified environment." [363, p. 15]. In this part, the three major classes of *software* reliability assessment are presented (Section 10.4):

Black box reliability analysis (P. 111): Estimation of the software reliability based on failure observations from testing or operation. These approaches are called *black box* approaches because internal details of the software are not considered.

Software metric based reliability analysis (P. 115): Reliability evaluation based on the static analysis of the software (e.g., lines of code, number of statements, complexity) or its development process and conditions (e.g., developer experience, applied testing methods).

Architecture-based reliability analysis (P. 119): Evaluation of the software system reliability from software component reliabilities and the system architecture (the way the system is composed out of the components). These approaches are sometimes called *component-based reliability estimation* (CBRE), or *grey* or *white box* approaches.

Many concepts of software reliability engineering are adapted from the older and successful techniques of hardware reliability. The application of hardware dependability methods to software has to be done with care, since there are some fundamental differences in the nature of hardware and software, and its failure processes. Therefore, well-established hardware dependability concepts might perform differently (usually not very well) for software. It was even proposed that "hardware-motivated measures such as mttf, mtbf should not be used for software without justification" [306].

Today, software reliability engineering is a separate domain. Research on software reliability measurement (e.g., the work of Cheung [95], Littlewood [305], and

I. Eusgeld, F.C. Freiling, and R. Reussner (Eds.): Dependability Metrics, LNCS 4909, pp. 104–125, 2008.

Musa et al. [363]) addressed the characteristics of software reliability and adapted hardware reliability metrics. However, empirical evaluation is important before dependability concepts, derived from hardware-approaches, can be applied to software. For instance, such an empirical evaluation of component-based reliability estimation was presented by Krishnamurthy and Mathur [284].

Despite major advantages, software reliability assessment (with models such as the reliability growth models) is not powerful enough to address very high reliability demands (such as 10^{-9} of failure probability per hour) [308].

Software Faults Are Design Faults

The main difference between "hardware" and "software" failures is the underlying fault model. Traditionally, the largest part of hardware failures is considered as result from physical wearout or deterioration. Sooner or later, these *natural faults* [26], *will* introduce faults into hardware components and hence lead to failures.

Experience has shown, that these physical effects are well-described by exponential equations in the relation to time. Usage commonly accelerates the reliability decrease, but even unused hardware deteriorates. Physical separation and fault isolation (e.g., high-impedance electrical connections and optical couplers, such as applied by Wensley et al. [509]) made it possible to assume (approximately) statistical independence of the failure processes (of natural faults). The fact that this so-called *independence assumption* holds for physical faults, does not only highly reduce the complexity of the reliability models. Moreover, it makes the use of redundancy very effective in the context of hardware fault tolerance. Concepts, such as "hot" redundancy in combination with voting, or standby redundancy (reconfiguration upon failure detection), made it feasible to design systems with high hardware reliabilities.

Design faults are a different source for failures. They result mainly from human error in the development process or maintenance. Design faults will cause a failure under certain circumstances. The probability of the activation of a design fault is typically only usage dependent and time independent. By the increasing complexity of hardware systems, design faults become more and more an issue for hardware reliability measurement, so that "the division between hardware and software reliability is somewhat artificial" [362, p. 38].

Software is pure design [309] and consequently, software failures are caused by design faults [362, p. 38], [363, p. 7]. Note, the term "design" is used in a broad sense in software dependability and refers to all software development steps from the requirements to realization [294, p. 48]. Therefore, faults that are introduced during the implementation are also consided as design faults. In contrast to hardware, software can be perfect (i.e. fault-free). Unfortunately, it is usually not feasible to develop complex fault-free software, and even then, it is rarely feasible to guarantee that software is free of faults. Some formal methods can prove the correctness of software - this means it matches to a specification document. However, today's formal verification techniques are not designed for the application to large software systems such as consumer operation systems or word processors. Furthermore, correctness does not ensure reliability because the specification document itself can already be faulty. As it is not feasible to develop complex software systems free of faults and the absence of faults cannot

be guaranteed, the reliability of software needs to be evaluated in order to fullfill high dependability requirements.

The failure process of design faults is different from the one of ("hardware") natural faults. Obviously, copies of (normal) software will fail together, if executed with the same parameters. This shows that the independence assumption does not hold. More precisely, the failure probabilities of software copies are completely dependent. This makes many hardware fault tolerance principles ineffective for software. Instead of using redundant copies, software reliability can be improved by using design diversity. A common approach for this is the so called N-version programming (surveyed in Avižienis [25], introduced by Chen and Avižienis [94]). However, the reseach of Knight and Leveson [279] indicates, that design diversity is likely to be less effective for software than N-modular redundancy is in hardware reliability engineering.

Some studies have shown that for complex systems, the majority of failures are typically caused by software faults (see, for example, Gray [194]). Although software faults are design faults, their behaviour in dependable systems is similar to transient hardware faults. This is due to the stochastic of their activation conditions [193].

Software Usage Profiles

Littlewood and Strigini [309] state that software reliability has to be a probabilistic measure because the failure process, i.e. the way faults become active and cause failures, depends on the input sequence and operation conditions, and those cannot be predicted with absolute certainty. Human behaviour introduces uncertainty and hence probability into software reliability, although software usually fails in the same way for same operational conditions and same parameters. An additional reason to claim a probabilistic measure is that it is usually only possible to approximate the number of faults of complex software system.

To issue different ways of usage, the concepts of *user profiles* [95] and *operational profiles* [360, 363] are common for (black box or white box) software reliability measurement. These models use probabilities to weight different ways of software usage. Usage profiles can be used for hardware as well. For software designers, it is easy (and often practice) to include "excessive extra functionality" [309]. From this point of view, the weighting of service requests seems especially important for software.

Besides software usage, other context information might have to be included into reliability assessment. This is required because software reliability is more sensitive to differences in operational contexts than hardware reliability [309, p. 179]. In other words, a piece of software that was reliable in one environment, might be very unreliable in a slightly different one.

10.2 Common Measures and Metrics: What Do I Measure?

Many software reliability metrics differ from hardware reliability metrics primarily in the models that are used for the computation (Section 10.4). Hardware reliability metrics are usually time dependent. Although the failure behavior of (software) design faults depends on usage and not directly on time, software reliability is usually expressed in relation to time, as well. Only as intermediate result, some reliability models

use time-independent metrics such as the reliabilities of paths, scenarios, or execution runs. A major advantage of time dependent software reliability metrics is that they can be combined with hardware reliability metrics to estimate the system reliabiliy [363, p. 229]. For the evaluation of software design alternatives, time independent reliability metrics might be easier to compare.

For reasons of completeness, we repeat the relationships between the basic reliability metrics from Musa et al. [363, p. 228] (as said before, these are very similar to the hardware reliability metrics in Section 9.3, Page 65):

- Reliability $R(t)$:
$$R(t) = 1 - F(t) \tag{1}$$
- Failure probability $F(t)$:
$$F(t) = 1 - R(t) \tag{2}$$
- Failure density $f(t)$ (for $F(t)$ differentiable):
$$f(t) = \frac{dF(t)}{dt} \tag{3}$$
- Hazard rate $z(t)$ (also called conditional failure density):
$$z(t) = \frac{f(t)}{R(t)} \tag{4}$$
- Reliability $R(t)$ (derived from the hazard rate):
$$R(t) = exp[-\int_0^t z(x)dx] \tag{5}$$
- Mean time to failure (MTTF) $= \Theta$ (with t as operating time):
$$MTTF = \Theta = \int_0^\infty R(t)dt \tag{6}$$
- For clock time as approximation to execution time, $M(t)$ presents the random process of the number of failures experienced by time t, and $m(t)$ denotes the realisation of $M(t)$. The *mean value function*, which represents the expected number of failures at time t is given by:
$$\mu(t) = E[M(t)] \tag{7}$$
- *Failure intensity function* or *failure rate function*:
$$\lambda(t) = \frac{d\mu(t)}{dt} \tag{8}$$
- Note that the term "failure intensity" is used as a synonym for "failure rate" by foundational work in software reliability research (e.g., Musa et al. [363]). Musa [362] states that the term "failure intensity" was chosen to avoid common confusions between "failure rate" and "hazard rate".

Other relations between hardware and software reliabilities are:

- The *probability of failure per demand* can be suitable for terminating software. It is given by $1 - R$, with R as the reliability of a single execution [192].
- Availability related metrics such as downtime, uptime, or reboot time are more related to combined hardware-software-systems.
- Terms such as "*lifetime*" are less common in the context of software reliability.

Dependability Benchmarks

Performance benchmarks such as SPEC have become a powerful tool to evaluate and to compare performance of computer systems. This approach has not been adapted to dependability aspects until recently. Silva and Madeira [448] give an overview on the role of dependability benchmarks.

The objective of dependability benchmarks is to standardize ways how dependability of computer systems can be assessed. Since it is difficult to objectify dependability evaluation, an important part of the benchmark developing process is to set up an evaluation workflow that is accepted by a wide range of companies and customers of computer systems. Acceptance can be described by the attributes representativeness, usefulness and agreement.

The principle structure of a dependability benchmark is shown in Figure 1. In addition to a workload usually defined in performance benchmarks, there is a fault load which is basically a set of faults and stressful conditions, and there are measures that are related to dependability. The measurements of the benchmark can either be used directly in order to compare different systems or it can be used as input for dependability models (see Section 10.4) in order to derive dependability metrics that have a scope beyond the benchmark's measurements.

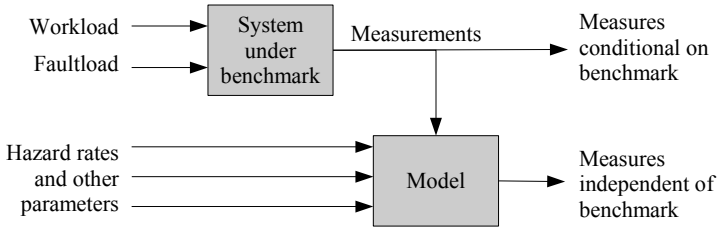

Fig. 1. Dependability Benchmarks

Silva and Madeira [448] also give references to dependability benchmarks that have been published recently.

10.3 Techniques for Measurement: What Data Is Necessary?

Just as a reminder, the title's question is worth repeating: What data is necessary? Data should not be collected only because it can be done. This would be just wasteful. First

of all a purpose, a goal should be defined that leads to questions that can be answered by collecting data. One method to achieve this is the GQM method described in Chapter 6.

The corresponding section on hardware reliability (s. Section 9.4) was divided into subsections on field data and fault injection among others. For software those terms have a slightly different meaning and significance. Furthermore, in the context of hardware reliability modeling, research and practice focus almost only on data about observed failures. For software the data used is much more diverse.

Program Size

Several models use the size or complexity of a program as input. A well-known metric for measuring program size is the *lines of code* metric (LOC) which is deceivingly simple. One problem with LOC is the ambiguity of the operational definition. Which lines are to be counted? Surely executable lines are counted, but what about two executable statements in one line? Lines containing data declarations only? Empty lines? Comments? Obviously, this problem can and has to be handled by a clear definition of LOC that is adhered to throughout the project.

Another problem is the obvious dependency of LOC on the programming language used which is typically a disturbing property in this context. An alternative measure for program size that abstracts from the programming language is the *function point* (FP). Developed in the late 1970s by Albrecht [16] function points basically are a weigted sum of the numbers of the following components of an application: external inputs, external outputs, user inquiries, logical internal files, and external interface files. This weighted sum is refined by the estimated complexity of those components and furthermore by 14 weighted general system characteristics. As FPs thus rely much more on the functional requirements of an application and not on the implementation, FPs are much more useful for doing comparisons across different programming languages and also across different companies. A common metric involving FPs, e.g., is "defects per FP".

Test Phase

Data collected during the test phase is often used to estimate the number of software faults remaining in a system which in turn often is used as input for reliability prediction. This estimation can either be done by looking at the numbers (and the rate) of faults found during testing [197] or just by looking at the effort that was spent on testing. The underlying assumption when looking at testing effort is "more testing leads to higher reliability". For example, Nagappan et al. [364], Nagappan [365], Nagappan et al. [366] evaluated the following metrics (and more) in this context:

- Number of test cases / source lines of code
- Number of test cases / number of requirements
- Test lines of code / sourcelines of code
- Number of assertions / source lines of code
- Number of test classes / number of source classes
- Number of conditionals/ number of source lines of code
- Number of lines of code / number of classes

Failure Data

Of course, information about observed failures can also be used for software reliability assessment. Data collected includes, e.g., date of occurence, nature of failures, consequences, fault types, and fault location [266].

In the case that field data is not available and testing does not yield a sufficient amount of failure data, *fault injection* can be applied. An introduction is given in Chapter 9.4. Fault models for software faults exist but are not as common as hardware fault models, yet. A well-known example is Orthogonal Defect Classification (ODC) [97]. It divides software faults in six groups: *assignment*, *checking*, *timing*, *algorithm*, and *function*. For emulation by an injector, these faults have to be "generated", which means that even if there is no fault in the code, the code is changed. For example, if a checking fault should be generated, a check in the code could be changed such that a less-or-equal check is replaced by a less check. When the running program reaches the particular location in the code, a false check is performed resulting in a checking fault. Note, that the goal of fault injection is to acquire data about *failures* – not the data about the fault that was injected should be observed but the ability of the rest of the system to handle the fault. An implementation of a software fault injector was described by Durães and Madeira [137, 138].

Another use case for software fault injection not directly related to reliability is the assessment of test suites. The basic idea is to inject a number of faults into a system, run the corresponding test suite, and use the percentage of injected faults detected by the test suite as an indicator for the coverage achieved by the test suite.

10.4 Modeling: How Do I Model?

Although hardware and software reliability is similar, they have to deal with failure rates of diverse characteristics. Under the assumption that the program code is not altered and the usage profile stays constant, software lacks the typical wear-out phase where failure rates rapidly increase after a long time of being quasi-constant (see Figure 4 in Chapter 9). However, the assumption that the code stays the same for the lifetime of a system does not hold. Typically, a software is under permanent development, testing and bug fixing. This affects failure rates in several ways. Smaller updates reduce the failure rate in most cases, except for those where the fix of one bug introduced others increasing the failure rate. On the other hand, the majority of software offers major updates from time to time that offer a bunch of new functionality introducing a lot of code that shows high failure rates. This often leads to jumps in the overall failure rate. Figure 2 sketches the effect.

A bunch of models have been developed trying to get a grip on the specifics of software failure rates. Some of the models will be introduced in the following sections. They are grouped by the amount of internal knowledge about the software and its structure. Black box reliability models do not rely on internal specifics of the software. Another group of models builds on software metrics such as complexity measures and a third group analyzes the internal structure of the software under consideration.

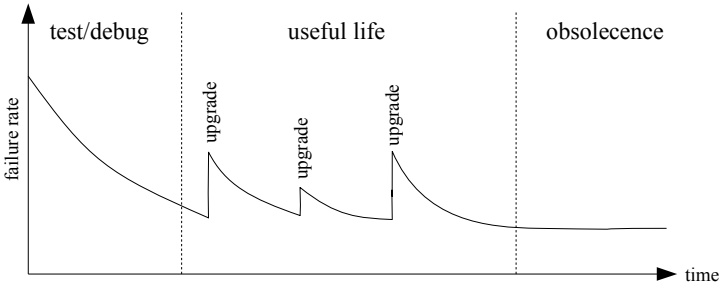

Fig. 2. A rough sketch of software failure rate over lifetime

Black Box Reliability Models

Software reliability estimation with black box models dates back to the year 1967 when Hudson [225] modeled program errors as a stochastic birth and death process. In the following years, a lot of models have been developed building on various stochastic properties. In their book "Software Reliability", Musa et al. [363] introduce a more general formalism that is able to capture most of the models that have been published. Farr [154] reiterates the overview of Musa et al. [363] but focuses more on the explicit description of each of the reliability models. The classification scheme of Musa et al. [363] groups software reliability models in terms of five attributes:

1. *Time domain:* Is the time base for the model calendar time or execution time?
2. *Category:* Is the number of failures that can be experienced in infinite time finite or infinite?
3. *Type:* What is the distribution of the number of failures experienced by time t?
4. *Class (for finite category only):* What is the functional form of the failure intesity in terms of time?
5. *Family (for infinite category only):* What is the functional form of the failure intensity in terms of the expected number of failures experienced?

The objective of this section is to sketch the major attributes in order to give an impression what properties are addressed by the attributes. A small set of well-known models will be described later in this section.

Time Domain. Musa [361] introduced a new notion of reliability modeling that was based on a software's execution time rather than calendar time. Times between failures are expressed in terms of computational processing units aiming at incorporating the stress induced on the software. Since execution time seems to be rather arbitrary for project managers, Musa added a second model component that relates execution time to calendar time by expenditures for human and computational resources.

Finite and Infinite Category. The *category* attribute classifies software reliability models according to the property whether the number of encountered failures tends to infinity or not in infinite time. Sounding rather theoretical, it classifies whether the software under consideration tends to be fault-free in infinite time or not. For example, if correction of a fault leads to other faults, the software may never be fault-free.

Poisson and Binomial Types. The distribution of the number of failures experienced by time t plays a major role in the classification of software reliabiliy models. The following section discusses both types in more details.

Distribution Class and Family. Reliability models of the finite and infinite category can each be subclassified according to the functional form of failure intensity modeling. *Failure intensity* is the number of failures per time unit. For models of the finite category, the functional form of failure intensity is described in terms of *time* by a distribution of a certain *class*. As models of the infinite category require an description of failure intensity in terms of the *expected number of failures*, it is described by a distribution of a certain *family*.

The intention of this section is not to provide a comprehensive overview of existing software reliability models but to sketch the basic ideas and to give some reference to the most well-known models.

Poisson and Binomial Type Models. Musa et al. [363] identified two types of models that differ in the underlying failure process. Whereas binomial type models assume that there is an initial number of faults u_0 in the program, Poisson-type models assume the initial number of faults to be a random variable with mean ω_0.

Binomial type models. Assume that there is a one-to-one correspondence between fault and failure. After each failure, the causing fault is repaired instantaneously and repair is perfect, which means that repair eliminates the problem and does not cause new ones. This assumption leads to the notion that each fault in the software occurs exactly once and that it is independent of other faults. It is assumed that each fault/failure occurs randomly in time according to a per-fault hazard rate $z_a(t)$, which is assumed to be the same for all faults.

Since the hazard rate is defined as

$$z_a(t) = \frac{f_a(t)}{1 - F_a(t)} \tag{9}$$

where $F_a(t)$ is the cumulative distribution function of the random variable T_a denoting time to failure of fault a and $f_a(t)$ is its density. By solving the differential Equation 9 we obtain

$$F_a(t) = 1 - \exp\left[-\int_0^t z_a(x)dx\right] \tag{10}$$

By conditioning on time t' we have

$$F_a(t|t') = \frac{F_a(t) - F_a(t')}{1 - F_a(t')} = 1 - \exp\left[\int_{t'}^t z_a(x)dx\right] \tag{11}$$

The essential notion for binomial-type models is that due to the hazard rate, by time t each fault a is removed with probability $F_a(t)$ and remains in the software with probability $1 - F_a(t)$. Since there are u_0 faults at $t = 0$ the probability that m out of u_0 faults are removed until time t is the value of the binomial distribution

$$P[M(t) = m] = \binom{u_0}{m}\left[F_a(t)\right]^m\left[1 - F_a(t)\right]^{u_0 - m} \tag{12}$$

This is why models building on the above assumptions are of binomial type.

In order to obtain an equation for reliability, we need to determine the probability

$$P[T_i > t_i | T_{i-1} = t_{i-1}] \tag{13}$$

where T_i is the random variable of the time of i-th failure. It denotes the probability that the next failure i occurs at time t_i given that the last occurred at t_{i-1}. The fact that $i-1$ failures have occurred implies that only $u_0 - i + 1$ faults remain in the software yielding:

$$P[T_i > t_i | T_{i-1} = t_{i-1}] = [1 - F_a(t_i | t_{i-1})]^{u_0 - i + 1} \tag{14}$$

Using Equation 11 yields

$$P[T_i > t_i | T_{i-1} = t_{i-1}] = \exp\left[-(u_0 - i + 1) \int_{t_{i-1}}^{t_i} z_a(x) dx\right] \tag{15}$$

Replacing the absolute time t_i by the temporal difference δt_i, which is the time from failure $i-1$ to failure i, we obtain an equation for reliability, that is dependent on the number of remaining faults $(u_0 - i + 1)$ and the time of the last failure t_{i-1}:

$$R(\delta t_i | t_{i-1}) = \exp\left[-(u_0 - i + 1) \int_{t_{i-1}}^{t_{i-1} + \delta t_i} z_a(x) dx\right] \tag{16}$$

If the hazard rate $z_a(t)$ is constant then the integral and hence reliability are independent of t_{i-1}.

Poisson-type models. Assume that the initial number of faults in a software is not known as is the case with binomial type models, but rather is a Poisson random variable with mean ω_0. Therefore, u_0 is being replaced by the random variable $U(0)$ and Equation 12 is transformed into

$$P[M(t) = m] = \sum_{x=0}^{\infty} \binom{x}{m} \Big[F_a(t)\Big]^m \Big[1 - F_a(t)\Big]^{x-m} \frac{\omega_0^x}{x!} \exp(-\omega_0) \tag{17}$$

where the first part is the binomial distribution for an initial number of x faults and the second part is the poisson distribution, yielding the probability that there are actually x faults given the mean ω_0.

This equation can be transformed into

$$P[M(t) = m] = \frac{[\omega_0 F_a(t)]^m}{m!} \exp[-\omega_0 F_a(t)] \tag{18}$$

showing that the assumption of a Poisson distribution of the number of initial faults leads to a Poisson distribution for the number of failures that have occurred until time t, which equals the number of faults removed.

Comparison. The two types of models described above are obviously similar. Both models assume that the hazard rate are the same for all faults. The Bayesian model of Littlewood and Verrall (see below) gives up this assumption. Since for Poisson-type models the number of failures is a random variable, they are able to accomodate, in an approximate fashion, for imperfect debugging that eventually introduces new faults during repair actions.

Having a closer look at the characteristics of the hazard rate of the entire program (not to be mixed with hazard rate of the single faults), it can be observed that binomial-type models have discontinuous program hazard rates. Each time a failure occurs it is removed and the program hazard rate decreases discontinuously, which seems realistic since the correction of a bug causes an immediate decrease. Poisson-type models do not show this property. However, in a real environment failures are not repaired immediately but at some random time after failure which is an argument in favour of the Poisson approach.

Besides from the number of failures experienced until time t, which was denoted by $M(t)$, and reliability $R(\delta t_i|t_{i-1})$, other reliability metrics such as mean time to failure (MTTF) can be derived from the stochastic process.

A Brief Overview of Existing Models. In the equations above, neither the fault hazard rate $z_a(t)$ nor the distribution of the time to the next fault/failure $f_a(t)$ and $F_a(t)$ respectively, have been specified. This is where many of the models that have been proposed differ. Since many of the models share assumptions about the characteristic of the hazard rate, Musa et al. introduced the "class" attribute. For example, the models proposed by Jelinski and Moranda [256] or Shooman [440] belong to the class of binomial type models with exponential hazard rates while the model proposed by Schneidewind [431] is a Poisson-type model with exponential hazard rates. Other classes include Weibull, Pareto or gamma distributions.

One well-known model should not be forgotten, even if it leaves the sketched framework in various ways: the model proposed by Littlewood and Verrall [310]. The authors postulated that software reliability is correlated with the belief that a software works correctly leading to the consequence that reliability changes even if no failure occurs. Therefore, reliability increases within failure-free time intervalls and changes discontinuously at the time of failure occurrence. The model incorporates both the case of fault elimination and of introducing new faults. An additional assumption is that faults do not have equal impact on system reliability since some are more likely to be executed than others. Littlewood and Verrall use a Bayesian framework where the prior distribution is determined by past data (e.g., from previous projects) and the posterior incorporates past and current data. By this approach, both small updates including bug fixes as well as major upgrades that most commonly introduce new bugs can be modeled. As might have become visible, the model is very powerful covering a large variety of software projects, however, it is quite complex and more difficult to apply.

Fitting Black Box Reliability Models to Measurement Data. Brocklehurst and Littlewood [72] assessed the accuracy of some reliability models such as Jelinski-Moranda or Littlewood-Verrall based on industrial datasets and observed that the reliability prediction of the different models varied heavily. The authors also provided an overview

of several techniques, how the divergence of predictions and real data can be measured. The techniques will be reiterated shortly, here.

From test data, two sets of data need to be extracted: Time to next failure and the model's reliability predictions. A straightforward way of comparison would be to take the predicted median time to failure and to count how many times the predicted median time was larger than the real time to next failure. If this is the case in approximately 50% of all predictions, the prediction could be valued accurate *in average*.

A more sophisticated approach is to draw a u-plot and to assess predictive accuracy in terms of divergence from the line of unit slope measured by, e.g., the Kolmogorov-Smirnov distance, which is the maximum vertical distance between both lines. Since the u-plot does not account for trends, a y-plot can be used instead of the u-plot.

The u-plot can be used to improve black box reliability models by fitting them to the MTTF values that are observed for a running system. The approach is also presented in Brocklehurst and Littlewood [72]: For the time between two successive occurrences of real failures, it is assumed that the cumulative reliability distribution estimated by the model $\hat{F}(t)$ can be linearly transformed by G such that $G[\hat{F}(t)]$ is equal to the true cumulative reliability distribution $F(t)$. Since G is also unknown, its estimate $G^\star$ is calculated by use of the u-plot obtained from previous observations: $G^\star$ is the polygon formed by successive u-plot steps. In Brocklehurst et al. [73] the same authors propose to replace the polygon by an SP-line yielding further improved prediction accuracy at the cost of more complex computations.

Software Metric Based Reliability Models

The objective is to reason about residual fault frequencies or failure frequencies which have to be expected when executing the software. Therefore, either static analysis of software using metrics such as lines of code, number of statements, or metrics measuring complexity can be used. On the other hand the development process and conditions under which software was developed influence its quality and such can also be used to estimate reliability.

Classification and Clustering Methods. The objective of *classification* methods is to learn how to assign data items to predefined classes. Clustering is the organization of data items into clusters based on similarity [248] without predefined classes.

A lot of research has been done and also is currently going on to investigate how classification and clustering methods can be used to assess the reliability of software and also hardware. For example Zhong et al. [526] describes how semi-supervised clustering is used to identify software modules as either fault-prone or not fault-prone. Classification methods are also useful to assess system reliability, e.g., Karunanithi et al. [268] use neural networks to predict the number of failures of a system after a given execution time based on time series information.

All classification and clustering methods have in common that the used data items are feature vectors $x = (x_1, ..., x_n)$ where x represents a single data item and every x_i with $i \in [1..n]$ is one measurement describing the data item, e.g., one could measure lines of code, number of methods and lines of comments for programs. This would result in one feature vector for each program. In principle every measurement described in Section 10.3 can be used as input data.

The usual procedure is that a set of data items–called training data–is used to train the clustering or classification algorithm. This phase is called training or learning phase of the algorithm. Afterwards the algorithm can be used to classify new unclassified data items, i.e., associate it with a class or cluster.

The literature distinguishes classification and clustering methods depending on the information used to train the algorithm:

Unsupervised: All clustering methods use unsupervised learning. Apart from the data collection and maybe depending on the algorithm the number K of clusters to be formed no information is available [195]. This only allows the partioning into clusters based on similarity and thus limits its usefulness for reliability assessment. Because of this unsupervised learning clustering is also called unsupervised classification [248].

Supervised: Supervised learning is required for classification. A data collection with additional knowledge about the data items, e.g., class labels is available for training.

Semi-supervised: A small amount of knowledge about the data collection is available, e.g., labels for some data items. The available data is not representative and thus cannot be used for a supervised algorithm [195].

There exist numerous algorithms for classification and clustering. For an introduction to clustering algorithms have a look at Jain et al. [248]. The current research dealing with classification of software or systems with respect to their reliability is using artifical neural networks as classification method. These have the advantage that they are able to develop the required model on their own in contrast to classical analytical models which have to be parametrized depending on the solved problem [269]. This parametrization is no trivial task. Karunanithi et al. [269] show that the neural nets which result from the training process are more complex than the usually used analytical methods by means of number of required parameters. Thus neural networks are easier to use and capture the problem complexity more accurate. For an introduction to artifical neural networks use Anderson and McNeil [18].

The most used approach for reliability assessement using classification is to take a set of data items somehow describing a program or a part of hardware and to label these data items with reliability information, e.g., number of residual faults or failure rates. This data collection is used to train a classification algorithm which later on is used to classify unknown software or hardware with respect to the used class labels. The following research follows this principle: Karunanithi et al. [268], Karunanithi et al. [269], Tian and Noore [469], Khoshgoftarr et al. [274], and Pai and Lin [387].

Karunanithi et al. [268], and Karunanithi et al. [269] were the first who used neural networks to realize a reliability growth prediction. As a training set pairs of execution times (input to the net) and observed fault counts (expected output) are used. These pairs represent the complete failure history of a system since the beginning of its testing up to some point. The trained net could be used to predict fault counts for future executions. Two types of prediction are distinguished: next-step and longterm prediction. The first predicts the output for the next point in a time series and the second predicts fault counts for some point in the future. Comparing the neural network approach to traditional analytical methods led to the observation that for longterm prediction the neural

network approach resulted in significant better predictions and for next-step prediction the results were insignificant less accurate.

Since these first approaches for reliability growth prediction many contributions in this direction were done: Some of the newer papers dealing with reliability growth prediction using neural networks are Tian and Noore [469], and Pai and Lin [387].

Neural networks could not only be used to predict failure/fault rates using time series data. Khoshgoftarr et al. [274] used a collection of classical software metrics such as number of lines of code, Halstead's effort metric, or McCabe's Cyclomatic complexity to determine how many faults are contained in a program. Since this approach does not consider environmental conditions such as problem complexity, and development environment the obtained results should be treated with caution (see Fenton and Ohlsson [159]).

The research described up to now uses supervised approaches for training the algorithms. Since this requires extensive labeled data collection to train the algorithm current research aims at using semi-supervised approaches. Seliya et al. [436] and Zhong et al. [526] describe an approach which uses clustering methods to partition data collections describing software modules. Afterwards an expert estimates for each cluster if the described software modules are fault-prone or not fault-prone. The assumption is that software modules within one cluster partition have similar properties with respect to fault-proness.

Bayesian Belief Nets. *Bayesian Belief Nets* (BBNs) are an old concept for graphically representing and reasoning about logical relationships between variables. It enables us to handle the uncertainty in the dependency between these variables by using conditional probabilities [447]. Reliability of software is mainly influenced by the quality of its development process which is very difficult to judge objectively. Thus, it is not possible to determine its influence with certainty. Fenton and Ohlsson [159] showed that to assess software quality more is required than using classical software metrics such as lines of code. They proposed the usage of BBNs to take further influences, e.g., quality of the development process, into account [157, 158]. Prinicpially, BBNs are also usable for assessing reliability of hardware. But since design faults are not the major issue with hardware, they are rarely used in this context.

Furthermore, BBNs are usable when other reliability prediction methods are not, because not enough data is available. For example in safety critical systems usually reliability growth models are not applicable, because the number of observed failures is far too low [71].

A BBN is a directed acyclic graph. Every node represents a discrete random variable,i.e. the predicate or statement which is represented by this variable is true or false with a certain probability. Edges represent causal dependencies between variables. For an example have a look at Figure 3.

Nodes which have only outgoing edges and no incoming ones are called root nodes. The variable represented by a root node is not influenced by any other variable. Nodes at the end of an outgoing edge are called children and the nodes with outgoing edges are parent nodes. The meaning is that children somehow depend on their parents. How a variable depends on other variables is determined by conditional probabilities. Every node with incoming edges has a node probability table (NPT) assigned to it. This table

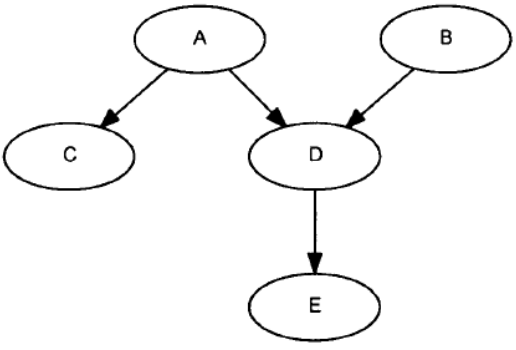

Fig. 3. Sample BBN [447]

contains conditional probabilities determining how a child depends on its parents. Root nodes are just assigned the probability for being true. Obviously, the probability for being false is the negation of the probability to be true.

To construct a BBN requires three stages [447]:

1. *Problem structuring:* In this step relevant variables are identified and the network structure is determined, i.e., the dependencies between the variables.
2. *Instantiation:* After defining the structure of the net, the probabilities have to be assigned. These may be derived from collected data or elicited from experts. For reliability predictions both is done. Amasaki et al. [17] extract the used probabilities from collected data, whereas Sigurdsson et al. [447] and Bouissou et al. [71] use expert knowledge.
3. *Inference:* In this step the net is evaluated using the baseyian theorm and theory of conditional probabilities. Known evidence about the state of variables is used to update the probabilities of the other variables and thus make statements about the probabilities of these variables becoming true or false. For example if we know that code reviews were made the probability that the software has no residual faults will increase. This statements are possible without using BBNs, but using BBNs makes them quantifiable, describes them more formal and prevents fallacies in reasoning due to misunderstanding of probability theory [71].

The main objective of BBNs is a what-if-analysis. On the one hand one can enter observed evidence, e.g., which tools are really used to improve design quality, and determine probabilities for all variables depending on this evidence. One of these variables usually will describe the reliability of the developed software, e.g. a predicate *residualFaultsExist*. This type of evaluation is called forward propagation.

On the other hand one could determine how big the influence of some variables onto others is. Thus, one can determine the benefit of methods such as code reviews or applied development models and use this knowledge to decide which methods are benefical and which not. This is called backward propagation since one first assumes that a reliable software was developed and with this evidence the conditional probability

$p(reliableSoftware|codeReview)$ can be computed,i.e. one goes from the dependen child node to its parent.

In reliability assessment BBNs are mostly used to model subjective knowledge about environmental conditions such as used development methods and tools, experience of developers and so on. For the first time this was done by Fenton and Neil [157, 158].

Advantages of BBNs in general are [447]:

- Easy to understand graphical representation.
- Combination of separate evidence sources.
- Easy to use.
- Takes uncertainty into account.
- Explicit modeling of dependencies.

In comparison to fault trees BBNs allow easier use of multistate variables, can model the undertainty in noisy gates and can capture sequentially dependent failures. In comparison to reliability block diagrams with BBNs common-cause failures can be modeled more naturally [447].

BBNs can also be used to complement already used mechanisms for predicting reliability. Amasaki et al. [17] observed that software reliability growth prediction sometimes predicts that a software is reliable, i.e. has few enough residual faults, for software which has no good quality at all. Thus, they proposed a solution where BBNs complement software reliability growth prediction by determining the probability that a software can be of high quality. For building the BBN the following data is used: product size, effort in the sense of person-day, detected faults, and residual faults.

For a deeper introduction into the theoretical foundations of BBNs refer to Pearl [391]. For learning how to use BBNs practically have a look at Jensen [259]. Bouissou et al. [71] gives a short less abstract introduction. Sigurdsson et al. [447] summarizes current work about using BBNs to represent expert knowledge in reliability assessment. The paper also gives advice how to obtain this expert knowledge.

Architecture-Based Reliability Models (White Box)

This subsection presents the basic approaches for reliability prediction of component-based software systems. Large software systems are often composed from smaller blocks that bundle functionality. In architecture-based reliability prediction, these blocks are named components. Without the need to refer to a special definition, components are just considered as basic entities of the software architecture. The architectural reliability models allow to predict the system reliability from the software architecture (containing components and connections between them) and the component reliability data.

The black box approaches, summarized above measure the reliability of a piece of software only based on observations from the outside. Intuitively, some software quality attributes, such as performance or reliability are compositional - the quality of a larger system seems to be derived from the quality of smaller parts and their relationship to each other. Architecture-based approaches follow this intuition by looking at the coarse-grained inner structure of software to measure the reliability.

A major advantage of architectural reliability (or performance) prediction approaches is that it is possible to predict the system reliability already early during the software design phase [441]. Failure data of the composed system is not required, as it is the case for the black box approaches. Therefore, potential quality problems might be discovered before a running system or prototype is implemented and black box approaches could be used.

The independence assumption is a major assumptions in reliability engineering. In the context of architectural reliability models, it assumes that the component reliabilities (as probability) are statistically independent. This allows to compute the reliability of a sequence of components as product of the component reliability. The independence assumption can lead to overly pessimistic reliability estimates, when the same components are executed multiple times in (large) loops.

The reuse of software components can affect the system reliability in both directions. Reuse of components plays a major role in making software development more effective. It is hoped to reduce development time and costs by reusing already developed and tested components. Reusing components can have a positive effect on the reliability of composite system because the components have already been part of a software product and taken part in its associated reliability growth. Therefore, the remaining number of failures might be smaller than that of newly developed components. However, reusing components can also be a reliability risk when some implicit assumptions about operation are not documented or ignored during reuse. As stated before, software (and therefore software components) is more sensitive to changes in their operational environment than hardware [309]. Software that has shown good reliability before, might perform bad in a slightly different context.

The following subsections are intended to provide a first idea on how the reliability of component-based software systems can be predicted, and which data is required. Surveys on architectural software reliability models have been published by Goševa-Popstojanova and Trivedi [192] and by Dimov and Punnekkat [120], focusing on the analysis of some more recent approaches. We limit this overview to provide simple examples for the three major classes, and describe only major conceptual extensions of later approaches.

We follow the structure of Goševa-Popstojanova and Trivedi [192], that distinguishes between three different classes of architecture based reliability models based on the way of combining the failure behaviour and the software architecture: state-based, path-based, or additive (using component reliabilities, omitting the architecture).

State-based Models. State-based approaches use probabilistic state representations, such as Markov chains, to model the transfer of control between components.

The early approach (for not continuously running applications) by Cheung [95, 96] uses a discrete time Markov chain (DTMC). It is created from a directed graph that represents the control flow between the software components. Without the loss of generality, N_1 is the single entry node and N_n is the single exit node. Matrix P contains the probabilities $P_{i,j}$ as possible transfer of control from node N_i to N_j.

As next step, the two absorbing states C and F are added, representing the states of correct and incorrect system output. This leads to a the set of nodes $\{C, F, N_1, \ldots, N_n\}$. Absorbing states have no outgoing transitions to other states.

Matrix $\hat{P}$ is derived from P with $\hat{P}(i,j)$ by including the probability R_i that a component i produces the correct result. As shown in Figure 10.4, direct transfers of control from a component back to itself are not allowed (except for C and F). Furthermore, N_1 (as start state) has no ingoing transactions, and when N_n is reached, there are no transfers of control back to the other nodes (except to C and F).

	C	F	N_1	N_2	$\dots$	N_j	$\dots$	N_n
C	**1**	**0**	0	0	$\dots$	0	$\dots$	0
F	0	**1**	0	0	$\dots$	0	$\dots$	0
N_1	0	$1-R_1$	**0**	R_iP_{12}	$\dots$	R_1P_{1j}	$\dots$	R_1P_{1n}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
N_i	0	$1-R_i$	**0**	R_iP_{i2}	$\dots$	R_iP_{ij}	$\dots$	R_iP_{in}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$
N_{n-1}	0	$1-R_{n-1}$	**0**	$R_{n-1}P_{(n-1)2}$	$\dots$	$R_{n-1}P_{(n-1)j}$	$\dots$	$R_{n-1}P_{(n-1)n}$
N_n	R_n	$1-R_n$	**0**	**0**	$\dots$	**0**	$\dots$	**0**

Fig. 4. Structure of the matrix $\hat{P}$ [95] as Markov model representation of correct transfer of control between components

$\hat{P}(i,j)$ represents only a single correct step in a execution sequence. The Markov Model has the nice property that $\hat{P}^n(i,j)$ denotes the probability of reaching the states $j \in \{C,F\}$ within n steps. Therefore, $\hat{P}^n(N_1,C)$ is the probability of correct termination in n or less steps.

Let the reduced matrix Q be created from $\hat{P}$ by removing the states C and F. The overall system reliability R is computed by

$$R = S(1,n)R_n \text{ ,with } S = I + Q + Q^2 + \cdots = \sum_{k=0}^{\infty} = (I-Q)^{-1} \qquad (19)$$

A variety of similar state-based models have been presented for terminating applications:

- The model of Kubat [285] uses *task-dependend* reliabilities for each component. These are derived from probabilistic component execution times. The component reliabilities are exponentially distributed with a constant failure rate.
- The failure rates in Gokhale et al. [186]'s model are time-dependent. Instead of using tasks (as Kubat [285]), different utilisation is modeled though the cumulative expected time spent in the component per execution.
- Hamlet et al. [207] use so-called *input profiles* as reliability parameter for each component. The input profiles are used to compute output profiles, which might be the input profiles for other components of the architecture.
- A similar parametrisation of usage is applied in the model of Reussner et al. [414], which computes the component reliability as a function of the usage profile. In addition, it enhances the approach of Hamlet et al. [207] by applying the idea of

parametrized contracts [413], which addresses the problem that functional and non-functional component properties highly depend on conditions of the deployment context. This idea is realized by making the properties of the services required (in the deployment context) a parameter of the component reliability function.

The reliability prediction of *continuously* running applications uses slightly different models. The required types of data are the exact or approximate distributions of the number of failures $N(t)$ in time interval $(0, t]$, the waiting time to first failure, and the failure rate [192]. The major approaches can be summarized as follows:

- Littlewood [305] models the architecture with an irreducible continuous time Markov chain (CTMC). A Poisson process predicts the occurrence of failures in the components using a constant failure rate.
- Ledoux [296] adds a second failure behaviour model to Laprie [292]'s approach. In addition to the failure behaviour model that assumes instantaneously restart, the second one addresses recovery times by delaying the execution for some time.
- Littlewood [307] generalises the earlier approach Littlewood [305] by characterising the system by an irreducible semi-Markov process (SMP).

Path-based Models. Path-based models abstract modularised system as paths. A path is understood as (mostly independent) sequence of components or statements.

Shooman [441] consider a path as a black box. This means that only f_i as the relative frequency of the execution of a path i of k total paths, and q_i as the probability of failure for a path are required. The number of failures to expect n_f in N system executions (= path runs) is given by

$$n_f = Nf_1q_1 + Nf_2q_2 + \ldots + Nf_kq_k = N\sum_{i=1}^{k} f_iq_i \tag{20}$$

For the number of paths N approaching infinity, the probability of failure of a execution run is given by

$$q_0 = \lim_{N\to\infty} \frac{n_f}{N} = \sum_{i=1}^{k} f_iq_i \tag{21}$$

Given the execution time t_i, assuming a rectangular time to failure distribution for a path ($t_i/2$ hours in average), the average system failure rate z_0 is computed as

$$z_0 = \frac{\sum_{i=1}^{k} f_iq_i}{\sum_{i=1}^{k} f_i(1-\frac{q_i}{2})t_i} = \frac{q_0}{\sum_{i=1}^{k} f_i(1-\frac{q_i}{2})t_i} \tag{22}$$

A similar path-based approach is presented and evaluated in Krishnamurthy and Mathur [284]. The concept of operational profiles [360] is used to generate a representative set of test cases T. This moves the weighting of b (f_i in the approach presented above) into an earlier step. The system reliability R can be computed as

$$R = \frac{\sum_{\forall t\in T} R_t}{|T|}, \tag{23}$$

where the reliability R^t of the path t is given by

$$R^t = \prod_{\forall m \in M(t)} R_m. \tag{24}$$

R_m denotes the reliability of a component m. $M(t)$ is the *component trace* of the test case t. A component trace can contain multiple invocations of the same component.

Krishnamurthy and Mathur [284] evaluate the problem of intra-component dependencies, which is the dependency between multiple invocations of the same component. The authors use an approach to "collapse" multiple occurrences of a component in a trace to a lower number. The degree of this process is referred as degree of independence. A DOI of ∞ means that no collapse is done at all. The DOI decreases the maximum number of component occurrences in component trace to k. For instance, the component trace t containing n executions of component j, the path reliability would be

$$R_c^t = R_j^{min(n,DOI)} \tag{25}$$

The work of Cortellessa et al. [109] is similar to the path-based approaches. Scenario-dependent component failure probabilities are derived from annotated UML Use-Case Diagrams and UML Sequence Diagrams. Component and connector failure rates are assumed to be known. The annotations of UML Deployment Diagrams allow the computation of component interactions probabilities between components in different deployment contexts. The failure probability of components and the one of the connectors failure probability are combined to determine the reliability of the whole system.

Additive Approaches. According to the characterisation of Goševa-Popstojanova and Trivedi [192], additive approaches are not explicitly using the software architecture, but still base the computation on component failure data. It is assumed that the component reliabilities can be modeled as nonhomogeneous Poisson Process (NHPP). Thus, the times between the occurrence of component failures are independent. The presented models combine the NHPPs of the components to a single NHPP for the system.

Assuming parallel component testing, Xie and Wohlin [514] estimate component reliabilities from component failure data (from independent testing). As the system is considered as a series system, every component failure leads to a system failure. This is a pessimistic assumption for fault tolerant system designs, because these might not show a system failure for every component failure. An other major assumption of additive models requires that the time t is set to zero for all components at the same time. This requires the introduction of the components at the same time point. This assumption allows to compute the system failure rate $\lambda_s(t)$ at time t simply by summing up the subsystem (component) failure rates $\lambda_i(t)$

$$\lambda_s(t) = \lambda_1(t) + \lambda_2(t) + \ldots + \lambda_n(t). \tag{26}$$

The corresponding cumulative number of system failures $\mu_s(t)$ (also known as mean value function) at time t is

$$\mu_s(t) = \sum_{i=1}^{n} u_i(t) = \int_0^t (\sum_{i=1}^{n} \lambda_i(s))ds. \tag{27}$$

A similar approach is presented by Everett [152]. It is argued, that the approach can be used before the system testing starts, because system failure data is not required for first predictions. Everett's model [152] differs from Xie and Wohlin [514] in determining the component reliabilities by an approach called Extended Execution Time (EET) model (see Everett [151]). The EET is in parts identical to Musa et al. [363]'s Basic Execution Time (BET) model. Both estimate component reliabilities from product and process properties such as the fault-density (e.g., estimated from earlier projects), lines of code, and other program and performance metrics. The EET extends the BET in using an additional parameter for modelling varying probabilities for the execution of instructions. For certain parameter values, the EET (and the BET) is a NHPP to model failure occurrence. This simplifies the computation, because the NHPP model allows to compute the cumulative failure rate function as sum of the corresponding component functions (as in Xie and Wohlin [514]'s Equations 26 and 27).

10.5 Proactive Schemes

In 1995, a new dependability technique called "rejuvenation" attracted attention, which is essentially a special preventive maintenance technique that tries to set components back to a "healthy", "young" state before they fail [224]. This approach differs from traditional fault tolerance techniques in that the system is not reacting to faults that have occurred but rather is trying to proactively deal with them, and it differs from traditional preventive maintenance approaches where maintenance implies manual check or replacement of field replaceable units. In the early 2000s, after publication of articles by Tennenhouse [467] and the autonomic computing manifesto by Horn [217], the new direction of proactive dependable computing gained importance.

In contrast to the attention that the topic attracted in terms of technical implementation, it is not thoroughly covered by dependability metrics. One exception is Garg et al. [179] where the authors modeled a system with rejuvenation by a Markov Regenerative Stochastic Petri Net (MRSPN) and solve it by use of a Markov Regenerative Process (MRGP) yielding the probability of being in an unavailable state. In 2005, Salfner and Malek [424] proposed a more general method to assess availability of systems employing proactive fault handling schemes. The proposed approach divides proactive fault handling into two parts: the *prediction* of upcoming failures and the *action* that the system performs upon prediction in order to deal with the fault. Two types of actions can be identified: Preventive actions try to prevent the occurrence of failures but also traditional repair actions can benefit from failure prediction by preparing for the upcoming failure in order to reduce time to repair. The accuracy of failure prediction, the effectiveness of countermeasures and the risk of introducing additional failures that would not have occurred without proactive fault handling are taken into account in order to compute the change in availability. However, the paper presented only a formula for steady-state availability. Metrics or models that cover other dependability metrics such as reliability are not available, yet.

10.6 Summary

This chapter covers dependability metrics that refer to software reliability. Although software and hardware reliability are related, several characteristics in which they differ are identified and discussed. Following the "goal-question-metric" paradigm, proper data collection is addressed including issues of testing, failure data and program attributes. The main focus is on models for software reliability assessment; the approaches are presented include black-box reliability models, reliability models that are based on other software metrics and white-box reliability models that build on knowledge about the internal structure of the system. For each group several models are described and compared. A brief survey on proactive schemes concludes the chapter.