26 Prognose von Softwarezuverlässigkeit, Softwareversagensfällen und Softwarefehlern

Michael Grottke

26.1 Einleitung

Im Laufe der letzten Jahrzehnte hat Software nicht nur in der Form von Computerprogrammen an Bedeutung gewonnen, sondern zudem als integraler Bestandteil von so unterschiedlichen Systemen wie Automobilen und medizinischen Geräten fast jeden Lebensbereich erobert. Diese Entwicklung hat zweifelsohne die Möglichkeiten der betroffenen Systeme erhöht. Allerdings ist der Mensch durch den Siegeszug der Software auch in zunehmendem Maße von ihrem korrekten Verhalten abhängig geworden.

Während die Unterstützung der Analyse-, Design- und Implementierungsphase der Softwareerstellung durch systematische Methoden und Werkzeuge gewisse Irrtümer von Softwareentwicklern verhindern kann, lässt sich eine völlige Programmkorrektheit praktisch nicht garantieren. Vielmehr ist immer damit zu rechnen, dass eine Software *Fehler* (insbesondere falsche oder fehlende Programmzeilen) aufweist, deren Aktivierung bei der Programmausführung ein *Versagen* zur Folge hat, also ein Softwareverhalten, welches von dem eigentlich spezifizierten abweicht. Unter *Softwarezuverlässigkeit* versteht man nun allgemein die Wahrscheinlichkeit dafür, dass in einer definierten Umgebung die Software innerhalb einer bestimmten Nutzungsperiode nicht versagt [3]. Bei kontinuierlich laufender Software wird die Länge der „Nutzungsperiode“ in der Regel in Form der zeitlichen Dauer der Programmnutzung gemessen [46]. Hingegen ist es bei Transaktionssystemen und Ähnlichem sinnvoll, die „Nutzungsperiode“ an der Zahl der Programmläufe festzumachen [3]. Da die beobachteten Versagensfälle dynamische Phänomene sind, deren Frequenz z. B. davon abhängt, wie häufig fehlerhafte Codestellen aufgerufen werden, bezieht sich ein geschätzter Zuverlässigkeitswert immer auf eine bestimmte Art der Programmnutzung. Obige Zuverlässigkeitsdefinition betont dies durch den expliziten Hinweis auf die „definierte Umgebung“.

Dieses Kapitel bietet einen Überblick über Modelle zur Prognose von Softwarezuverlässigkeit, Softwareversagensfällen und Softwarefehlern. Mit den in Abschn. 26.2 behandelten Modellen kann man aus dem Versagensverlauf während der Testphase die Zuverlässigkeit im Nutzungsbetrieb oder die bis zum Testende zu erwartenden Versagensfälle vorhersagen. Abschnitt 26.3 enthält knappe Darstellungen einiger weiterer Modellklassen: In Abschn. 26.3.1 sind Modelle zusammengefasst, die basierend auf ein oder zwei Test-Stichproben die Zuverlässigkeit der Software prognostizieren oder ihren Fehlergehalt abschätzen. Demgegenüber benötigen die in Abschn. 26.3.2 beschriebenen Modelle keine Beobachtungen aus der Testphase; sie versuchen, aufgrund von Informationen über das Softwareprodukt und seine Erstellung die Zahl der Softwarefehler vorherzusagen. Dass in dem vorgegebenen Rahmen die Fragestellungen nur angerissen und die Methoden nur skizziert werden können, liegt auf der Hand. Der interessierte Leser sei deshalb auf weitere Überblicksartikel [3], [5], [13], [17], [18], [24], [32], [40], [54], [57], [64], [65] sowie auf Bücher [4], [10], [21], [36], [45], [47], [52], [58], [63] zum Themenkomplex verwiesen.

In P. Mertens und S. Rässler (Hrsg.), *Prognoserechnung*, 7. Auflage, Springer/Physica, 2012, S. 585–619.

26.2 Softwarezuverlässigkeitswachstumsmodelle

Die in diesem Abschnitt vorgestellten Modelle betrachten allesamt, wie sich die Anzahl der beobachteten Versagensfälle im Laufe der Integrations- oder Systemtestphase entwickelt. Während des Testens wird das Versagen der Software zum Anlass genommen, die ursächlichen Fehler im Code aufzuspüren und zu korrigieren, sodass die Zuverlässigkeit der Software sich im Zeitablauf verändert und dabei tendenziell zunimmt. Da die hier beschriebenen Modelle versuchen, diesen Effekt abzubilden, werden sie als „Softwarezuverlässigkeitswachstumsmodelle“ (kurz: SZWM) bezeichnet. SZWM wurden unter anderem bei der Entwicklung von Teilen der Space-Shuttle-Software erfolgreich eingesetzt [42], [55].

Zur Anwendung der Modelle benötigt man entweder die Zeitpunkte, zu denen ein Softwareversagen beobachtet wurde, oder aber für eine Menge von Zeitintervallen, welche den gesamten Beobachtungszeitraum partitionieren, die jeweilige Zahl der in ihnen aufgetretenen Versagensfälle. Hierbei kann das verwendete Zeitmaß im Prinzip auch die Kalenderzeit sein. Allerdings gehen die meisten der im Folgenden diskutierten Modelle davon aus, dass die Intensität der Programmnutzung im Zeitablauf konstant ist. Deshalb sollte ein Maß zugrunde gelegt werden, für welches dies in etwa zutrifft, z. B. die von den Testern zur Durchführung der Testfälle benötigte Zeit oder die CPU-Ausführungszeit.

Obwohl Software deterministisch reagiert – auf exakt identische Eingabewerte unter denselben Nebenbedingungen also immer das gleiche Ergebnis produziert – ist es aus den folgenden Gründen dennoch sinnvoll, die Versagenszeitpunkte als zufällig aufzufassen [45], S. 29 f.:

1. Die Irrtümer seitens der Softwareentwickler, die zur Einbringung von Fehlern in das Programm führen, sind nicht mit Sicherheit vorhersehbar. Deshalb sind die Positionen der Fehler in der Software unbekannt.

2. Selbst wenn ein bestimmtes Nutzungsprofil mit vorgegebenen Frequenzen für die Aufrufe der einzelnen Funktionsbereiche zugrunde gelegt wird, ist die exakte Sequenz der Nutzereingaben nicht von vornherein festgelegt.

Die Zufallsvariablen $T_1 < T_2 < ...$, also die Zeitpunkte, zu denen das erste, zweite, ... Versagen auftritt, können mit einem stochastischen Punktprozess modelliert werden. Bezeichnet man mit $M(t)$ die Anzahl der Versagensfälle im Intervall $(0, t]$, so handelt es sich bei $\{M(t),\ t \geq 0\}$ um einen mit dem Punktprozess verbundenen stochastischen Zählprozess. Ein solcher zeichnet sich dadurch aus, dass er nur null oder ganzzahlige positive Werte annehmen kann und im Zeitablauf nichtabnehmend ist. Verschiedene Zählprozesse unterscheiden sich darin, wie der Zuwachs in der Anzahl der beobachteten Ereignisse erfolgen kann. Stellt man sich die Wertausprägungen $0, 1, 2, ...$ als mögliche Zustände des Prozesses vor, so läuft die Frage darauf hinaus, wie die Übergänge zwischen diesen Zuständen spezifiziert sind.

Eine große Gruppe von Zählprozessen geht davon aus, dass nicht mehr als ein Ereignis gleichzeitig eintreten kann. Somit ist von einem beliebigen Zustand j direkt nur der Zustand $j+1$ erreichbar. Für ein kurzes Zeitintervall $(t, t+\Delta t]$ kann man plausiblerweise annehmen, dass die Wahrscheinlichkeit für einen Wechsel vom Zustand j in den Zu-

stand $j+1$ proportional zur Intervalllänge Δt ist. Im einfachsten Fall ergibt sich die Übergangswahrscheinlichkeit als Produkt von Δt mit einer Konstanten z. Da in unserem Kontext ein höherer Wert dieser Konstante für jeden Moment in dem Zeitintervall eine größere augenblickliche Gefahr eines Versagenseintritts bedeutet, wird z als Programmhazardrate bezeichnet.

Gaudoin [15], S. 37 ff., und später Chen und Singpurwalla [11] haben gezeigt, dass sich viele SZWM als Spezialfälle des so genannten selbstanregenden Punktprozesses (*self-exciting point process*) darstellen lassen. In diesem komplizierteren Punktprozess ist die Programmhazardrate kein konstanter Wert, sondern sie kann sowohl vom aktuellen Zeitpunkt t abhängen als auch von der gesamten Vorgeschichte des Zählprozesses, nämlich der Anzahl der bisherigen Versagensfälle $M(t)$ und der Menge der Versagenszeitpunkte $D_t = \{T_1, T_2, ..., T_{M(t)}\}$, falls $M(t) \geq 1$, bzw. $D_t = \varnothing$, falls $M(t) = 0$. Formal ist die Programmhazardrate wie folgt definiert (vgl. [15], S. 39, und [59], S. 289):

$$Z(t, M(t), D_t) = \lim_{\Delta t \to 0} \frac{P(M(t+\Delta t) - M(t) = 1 \mid M(t), D_t)}{\Delta t}.$$

Da die Programmhazardrate beim selbstanregenden Punktprozess eine Funktion der Zufallsvariablen $M(t)$ sowie der Zufallsvariablen in der Menge D_t sein kann, ist sie selbst potenziell stochastisch. Erst wenn feststeht, dass die Software bis zum Zeitpunkt t insgesamt $m(t)$-mal versagt hat und die Menge der tatsächlichen Versagenszeitpunkte $d_t = \{t_1, t_2, ..., t_{m(t)}\}$ beträgt, ist ihre Realisation zum Zeitpunkt t, bezeichnet als $z(t,m(t),d_t)$, auf jeden Fall determiniert. (Kleinbuchstaben repräsentieren hierbei jeweils die Realisationen der korrespondierenden Zufallsvariablen.) Betrachtet man die Realisation der Programmhazardrate im Zeitablauf, so ändert sich diese aufgrund ihrer Abhängigkeit von $m(t)$ bei vielen SZWM mit jedem Versagenseintritt abrupt, während sie zwischen den Versagenszeitpunkten gar nicht oder nur kontinuierlich variiert. Wegen dieses oft stückweisen Aufbaus wird die Programmhazardrate mitunter auch „verkettete Versagensrate“ [11] genannt.

Die Annahmen des selbstanregenden Punktprozesses im Kontext eines SZWM können informell folgendermaßen beschrieben werden (vgl. [24], [32]):

1. Zum Zeitpunkt $t = 0$ ist noch kein Versagensfall eingetreten, d. h. $M(0) = 0$.

2. Falls die Realisationen $m(t)$ und d_t feststehen, entspricht die Wahrscheinlichkeit für genau ein Softwareversagen in dem Zeitintervall $(t, t+\Delta t]$ annähernd dem Produkt aus dessen Länge Δt und der Realisation der Programmhazardrate $z(t,m(t),d_t)$.

3. Die Wahrscheinlichkeit für das Auftreten von mehr als einem Versagensfall in dem Intervall $(t, t+\Delta t]$ geht mit $\Delta t \to 0$ schneller gegen null als die Wahrscheinlichkeit für das Auftreten von genau einem Versagensfall. Praktisch bedeutet dies, dass in einem sehr kurzen Zeitintervall nicht mehr als ein Versagensfall beobachtet werden kann.

Die Struktur dieses allgemeinen selbstanregenden Punktprozesses ist in Abb. 26.1 dargestellt. Hierbei entsprechen die Kreise den Zuständen des Zählprozesses $M(t)$, von denen jeweils nur ein Übergang in den nächsthöheren Zustand möglich ist. Die Grafik deutet an, dass sich gemäß mancher Modelle maximal u_0 Versagensfälle einstellen können und somit die Zustände $u_0+1, u_0+2, ...$ nicht existieren.

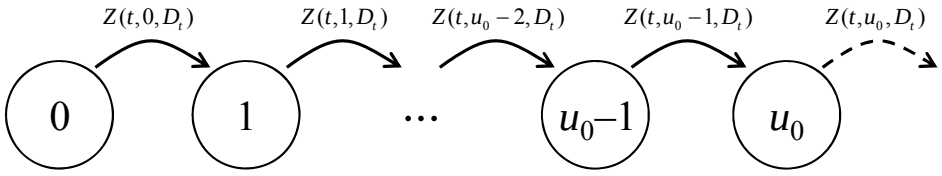

Abb. 26.1 Struktur der Anzahl der Versagensfälle als selbstanregender Punktprozess

Den Erwartungswert der Programmhazardrate bezüglich $M(t)$ und D_t, also denjenigen Wert, welcher ohne Kenntnis der Vorgeschichte des Zählprozesses zum Zeitpunkt t für die Programmhazardrate erwartet wird, bezeichnet man als Versagensintensität $\lambda(t)$:

$$\lambda(t) = E\left[Z(t, M(t), D_t)\right].$$

Integriert man die Versagensintensität von null bis t, so erhält man die Mittelwertfunktion $\mu(t)$, welche die erwartete Anzahl an Versagensfällen im Intervall $(0, t]$ angibt:

$$\mu(t) = \int_0^t \lambda(y)\, dy = \sum_{j=0}^{\infty} j \cdot P(M(t) = j) = E[M(t)].$$

Die grafische Darstellung von Mittelwertfunktion und Versagensintensität eines SZWM erlaubt einen guten Eindruck davon, wie sich aufgrund der Modellannahmen die Versagensauftritte erwartungsgemäß über den Beobachtungszeitraum verteilen.

Falls man davon ausgehen kann, dass sich die Programmhazardrate weiter wie vom Modell spezifiziert entwickeln wird, lässt sich mit ihrer Hilfe auch die zukünftige Zuverlässigkeit der Software bestimmen. Bedingt auf die Realisationen $m(t)$ und d_t beträgt die Wahrscheinlichkeit für kein Versagen im Zeitintervall $(t, t+\Delta t]$:

$$R(\Delta t \mid t, m(t), d_t) = \exp\left(-\int_t^{t+\Delta t} z(y, m(t), d_t)\, dy\right). \tag{1}$$

Die im Folgenden dargestellten SZWM sind alle Spezialfälle des selbstanregenden Punktprozesses. Ihre Gruppierung erfolgt unter dem Gesichtspunkt, welche Teile der aus $M(t)$ und D_t bestehenden Vorgeschichte des Zählprozesses annahmegemäß die Programmhazardrate beeinflussen.

26.2.1 Markovprozess-Modelle mit Einfluss des aktuellen Zustands

Die während des Testens aufgetretenen Versagensfälle geben den Anstoß zur Beseitigung der ursächlichen Fehler. Eine idealisierte Modellierung des Testprozesses geht davon aus, dass jede Fehlerkorrektur unmittelbar nach dem beobachteten Versagen erfolgt und erst im Anschluss daran die Testausführung und die Zeitnahme fortgesetzt werden. (Obwohl dies in den allerwenigsten Fällen der Realität entspricht, kann man sich dieser modellhaften Situation annähern, indem man für jeden Fehler nur das erste von ihm bewirkte Versagen berücksichtigt und somit im weiteren Testverlauf so tut, als ob der Fehler nicht mehr im Code vorhanden wäre.) Unter diesen Voraussetzungen ist es plausibel anzunehmen, dass die Programmhazardrate von der Anzahl der bisherigen Versagensfälle $M(t)$ abhängt, da auf jeden Versagenseintritt eine sofortige

Änderung des Fehlergehalts der Software folgt. Bei den in diesem Abschnitt besprochenen SZWM nimmt zwar der aktuelle Prozesszustand $M(t)$ Einfluss auf die Programmhazardrate, die Menge der Versagenszeitpunkte D_t hingegen nicht:

$$Z(t, M(t), D_t) = Z(t, M(t)).$$

Diese Modelle stellen deshalb einen Sonderfall der Markovprozess-Modelle dar, bei welchen die Zukunft des Prozesses einzig von dessen aktuellem Zustand abhängen darf (aber nicht muss), nicht jedoch von weiteren Elementen seiner Vorgeschichte. Bei manchen der Modelle nimmt man zudem an, dass die Programmhazardrate auch von der momentanen Beobachtungszeit t selbst, die nicht Teil der Prozessvorgeschichte ist, abhängt. (Die hier behandelten SZWM sind alle zeitstetiger Natur; mit zeitdiskreten Markovprozess-Modellen befasst sich Kap. 14 dieses Buches.)

Eine besondere Unterklasse bilden die Binomialmodelle (siehe Musa und andere [47], Kapitel 10.3 und 11.1.1, sowie Shantikumar [56]). Diese SZWM gehen davon aus, dass sich zu Testbeginn eine bestimmte Zahl von Fehlern, u_0, in der Software befindet. Zudem unterstellen sie, dass hinsichtlich ihrer Tendenz, ein Versagen zu verursachen, alle Softwarefehler zu jedem Zeitpunkt gleich gefährlich sind. (Wie hier beziehen wir auch im Folgenden die „Gefährlichkeit" eines Fehlers einzig auf seine Äußerungsrate und nicht auf den im Falle seiner Aktivierung verursachten Schaden.) Jeder Fehler weist also die gleiche Hazardrate $z_a(t)$ auf, die zwar von der Zeit, nicht aber von der Prozessvorgeschichte abhängen kann und daher nicht-stochastisch ist. Somit ist für jeden einzelnen Fehler die Wahrscheinlichkeit dafür, bis zum Zeitpunkt t ein Versagen bewirkt zu haben, identisch

$$F_a(t) = 1 - \exp\left(-\int_0^t z_a(y)\, dy\right).$$

Da die Binomialmodelle zudem davon ausgehen, dass die Fehlerkorrektur perfekt erfolgt, ergibt sich die Programmhazardrate zu jedem Zeitpunkt als Produkt der Anzahl der noch verbliebenen Fehler, $u_0 - M(t)$, mit dem Beitrag eines einzelnen Fehlers, $z_a(t)$:

$$Z(t, M(t)) = (u_0 - M(t)) z_a(t)\,. \tag{2}$$

Nach dem u_0-ten Versagen und der Beseitigung des letzten Fehlers ist die Programmhazardrate gleich null. Deshalb kann der stochastische Zählprozess $\{M(t),\ t \geq 0\}$ bei Binomialmodellen die Zustände u_0+1, u_0+2, ... nicht annehmen.

Auch die Mittelwertfunktion lässt sich für die Binomialmodelle in intuitiv eingängiger Weise darstellen. Sie ist die Wahrscheinlichkeit eines jeden Fehlers, bereits ein Versagen verursacht zu haben, multipliziert mit der Zahl der ursprünglichen Softwarefehler:

$$\mu(t) = u_0 F_a(t) = u_0 \left[1 - \exp\left(-\int_0^t z_a(y)\, dy\right)\right]. \tag{3}$$

Wurde also z. B. innerhalb einer bestimmten Testdauer t jeder Fehler mit 20-prozentiger Wahrscheinlichkeit entdeckt, so kann man erwarten, dass zum Zeitpunkt t ein Fünftel der ursprünglich in der Software enthaltenen Fehler gefunden wurde.

Jelinski-Moranda-Modell

Das einfachste Binomialmodell wurde 1972 von Jelinski und Moranda [27] vorgeschlagen – als eines der ersten SZWM überhaupt. Es geht nicht nur davon aus, dass alle Fehler zu jedem Zeitpunkt gleich gefährlich sind, sondern unterstellt zudem eine konstante fehlerbezogene Hazardrate $z_a(t)$ von ϕ. Aus der für Binomialmodelle allgemein geltenden Gleichung (2) folgt deshalb für die Programmhazardrate:

$$Z(t, M(t)) = (u_0 - M(t)) z_a(t) = (u_0 - M(t))\phi .$$

Ihre Realisation $z(t,m(t))$ nimmt damit einen treppenförmigen Verlauf an: Sofort nach einem Versagensfall sinkt sie exakt um den Betrag ϕ (die Hazardrate des soeben entdeckten und korrigierten Fehlers) und bleibt dann bis zum nächsten Versagen konstant. Im linken Teil von Abb. 26.2 ist eine typische Realisation der Programmhazardrate dargestellt.

Für die Mittelwertfunktion ergibt sich aus Gleichung (3)

$$\mu(t) = u_0 \left[1 - \exp\left(-\int_0^t \phi \, dy \right) \right] = u_0 \left[1 - \exp(-\phi t) \right] .$$

Da sich dieser Ausdruck mit wachsendem t immer mehr an u_0 annähert, ist dem Modell gemäß zu erwarten, dass nach einer unendlichen Testdauer alle Softwarefehler gefunden und behoben sein werden. Eine für das Jelinski-Moranda-Modell typische Mittelwertfunktion und ihre Ableitung, die Versagensintensität

$$\lambda(t) = u_0 \phi \exp(-\phi t) ,$$

zeigt der rechte Teil von Abb. 26.2. Die Grafik macht noch einmal deutlich, dass die Versagensintensität als eine von der konkreten Prozessvorgeschichte unbeeinflusste *erwartete* Programmhazardrate eine *stetige* Funktion der Zeit ist. Beim Jelinski-Moranda-Modell nimmt sie im Zeitablauf kontinuierlich ab.

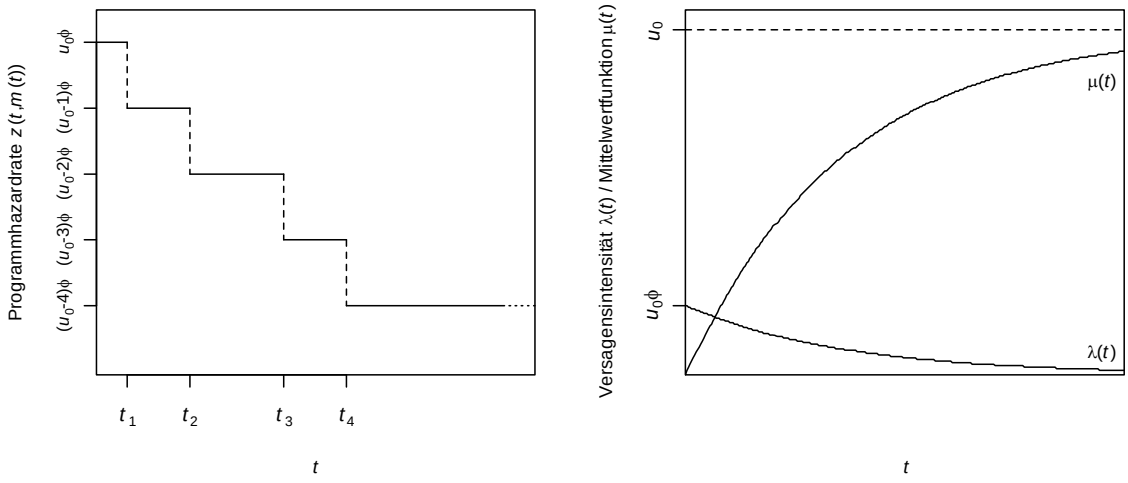

Abb. 26.2: Typische Realisation der Programmhazardrate, Versagensintensität und Mittelwertfunktion gemäß dem Jelinski-Moranda-Modell

Wurde die Software bis zum Zeitpunkt t getestet, wobei sie $m(t)$-mal versagte, so beträgt ihre Zuverlässigkeit im Intervall $(t, t+\Delta t]$ unter Anwendung von Gleichung (1)

$$R(\Delta t \mid t, m(t)) = \exp\left(-\int_t^{t+\Delta t} z(y, m(t))\, dy\right) = \exp\left(-(u_0 - m(t))\phi\Delta t\right).$$

Offensichtlich ist der Zuverlässigkeitswert (als Wahrscheinlichkeit einer versagensfreien Programmnutzung) umso geringer, je mehr Fehler sich noch in der Software befinden, je größer die Hazardrate eines einzelnen Fehlers ist und je länger die Zeitspanne Δt ausfällt, in welcher die Software verwendet wird.

Um für ein konkretes Softwareprodukt die Anzahl an Versagensfällen oder die Zuverlässigkeit prognostizieren zu können, müssen die beiden Parameter u_0 und ϕ geschätzt werden. Die Anwendung der Maximum-Likelihood-Methode führt zu einem System aus zwei nichtlinearen Gleichungen [13], welches leicht gelöst werden kann. Allerdings hat sich gezeigt, dass die so gewonnenen Schätzer unschöne Eigenschaften besitzen [14], [35]. Littlewood und Sofer entwickeln deshalb eine Bayes-Erweiterung des Jelinski-Moranda-Modells, deren Prognosequalität allerdings nur wenig besser ist (siehe [1]). Dies spricht dafür, dass die Probleme nicht nur vom Schätzverfahren, sondern auch von den simplen Modellannahmen selbst verursacht werden. Insbesondere die beiden folgenden Einwände werden oft erhoben:

1. Die Annahme der gleichen konstanten Hazardrate ϕ für alle Fehler ist unrealistisch. Tatsächlich ist es plausibler davon auszugehen, dass diejenigen Fehler, die früh im Test ein Versagen verursacht haben, im Hinblick auf ihre Äußerungsrate auch gefährlicher waren. Deshalb sollten die Sprünge, welche die Realisation der Programmhazardrate an den Versagenszeitpunkten macht, im Zeitablauf tendenziell abnehmen.

2. Obwohl nach jedem Versagensfall der Versuch unternommen wird, den für ihn kausalen Fehler zu korrigieren, gibt es in Wirklichkeit keine Gewähr dafür, dass die Bereinigung gelingt und keine neuen Fehler in die Software eingebracht werden. Deshalb könnte sich nach einem Versagenseintritt die Programmqualität sogar verschlechtern. Ein SZWM sollte diese Möglichkeit zulassen.

Viele der komplexeren Modelle entstanden als Antwort auf diese Kritikpunkte.

Moranda-Modell

So berücksichtigt z. B. Moranda [43] in seinem so genannten „geometrischen Modell" den ersten Einwand, indem er unterstellt, dass die Programmhazardrate nach jedem Softwareversagen auf das k-fache des vorherigen Wertes sinkt, wobei k zwischen null und eins liegt. Zu Testbeginn beträgt die Programmhazardrate ϕ. Es gilt also:

$$Z(t, M(t)) = \phi k^{M(t)}.$$

Wie die typische Realisation der Programmhazardrate im linken Teil von Abb. 26.3 zeigt, nimmt unter diesen Annahmen das Ausmaß, in welchem sich die Realisation der Programmhazardrate verringert, mit jeder Fehlerkorrektur ab.

Die Ermittlung von Mittelwertfunktion und Versagensintensität gestaltet sich für dieses Modell schwierig. Musa und andere [47], S. 572, leiten die Näherungen

$$\mu(t) \approx -\frac{1}{\ln(k)} \ln\left(1 - \frac{\phi}{k}\ln(k)t\right) \quad \text{und} \quad \lambda(t) \approx \frac{\phi}{k - \phi\ln(k)t}$$

her. Die exakten, aber unhandlichen Ausdrücke, deren Werte sich für bestimmte Wertkombinationen für ϕ und k deutlich von den Approximationen unterscheiden können, werden von Boland und Singh [6] angegeben.

Im rechten Teil von Abb. 26.3 sind typische Verläufe der approximierten Mittelwertfunktion und Versagensintensität angetragen. Im Vergleich mit dem Jelinski-Moranda-Modell wird deutlich, dass die Mittelwertfunktion im Zeitablauf nicht gegen einen festen Wert strebt, sondern die erwartete Anzahl an Versagensfällen beliebig groß werden kann, wenn die Testphase entsprechend lange dauert. Während es unplausibel erscheint, eine unendliche Zahl von ursprünglichen Softwarefehlern anzunehmen, liegt eine mögliche Interpretation dieser Mittelwertfunktion darin, dass die verminderte Wirkung der Korrekturen auf die Programmhazardrate nicht nur durch die geringere Gefährlichkeit der später entdeckten Fehler verursacht wird. Vielmehr werden zudem neue Fehler in die Software eingebracht, was in diesem Modell jedoch annahmegemäß in keinem Fall zu einem Anwachsen der Programmhazardrate führt.

Nachdem bis zum Zeitpunkt t insgesamt $m(t)$ Versagensfälle beobachtet wurden, errechnet sich die Zuverlässigkeit des Programms gemäß Gleichung (1) als

$$R(\Delta t \mid t, m(t)) = \exp\left(-\int_t^{t+\Delta t} z(y, m(t))\, dy\right) = \exp\left(-\phi k^{m(t)} \Delta t\right).$$

Auch im Moranda-Modell resultiert die Parameterschätzung nach der Maximum-Likelihood-Methode in einem System aus zwei gemeinsam zu lösenden Gleichungen [13].

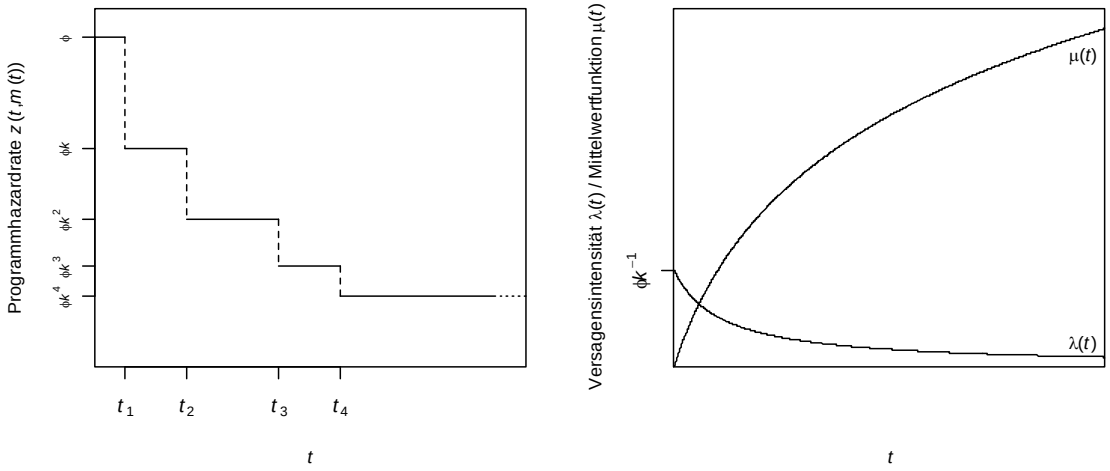

Abb. 26.3: Typische Realisation der Programmhazardrate, approximierte Versagensintensität und approximierte Mittelwertfunktion gemäß dem Moranda-Modell

Littlewood-Modell

Der nachlassende Effekt jeder weiteren Fehlerkorrektur auf die Programmhazardrate lässt sich auch im Rahmen eines Binomialmodells darstellen. An das Jelinski-Moranda-Modell anknüpfend geht Littlewood [33] davon aus, dass jeder Fehler zwar eine über die Zeit hinweg konstante Hazardrate ϕ besitzt; diese hat jedoch nicht für alle Fehler den gleichen fixen Wert. Stattdessen handelt es sich bei den Hazardraten der verschiedenen Fehler um Zufallszüge aus einer Gamma(α,β)-Verteilung. Während somit die Zeitspanne, nach der ein bestimmter Fehler ein Versagen auslöst, *bedingt* auf die gezogene Hazardrate ϕ einer Exponentialverteilung mit Parameter ϕ folgt, handelt es sich bei der *unbedingten* Verteilung dieser Zeitspanne um eine Pareto-Verteilung mit Verteilungsfunktion

$$F_a(t) = 1 - \left(\frac{\beta}{\beta + t} \right)^{\alpha} \tag{4}$$

und Hazardrate

$$z_a(t) = \frac{\alpha}{\beta + t}. \tag{5}$$

Diese im Zeitablauf abnehmende Hazardrate scheint im Widerspruch zu der postulierten konstanten Hazardrate ϕ zu stehen. Sie erklärt sich dadurch, dass ein Fehler mit hoher konstanter Hazardrate erwartungsgemäß früher zu einem Versagen führt. Im Umkehrschluss ist die Gefährlichkeit eines Fehlers tendenziell umso geringer, je länger er bereits unentdeckt geblieben ist. Genau dies spiegelt sich in Gleichung (5) wider.

Da es sich bei dem Littlewood-Modell um ein Binomialmodell handelt, ist die Programmhazardrate zum Zeitpunkt t das Produkt aus der Anzahl der noch verbliebenen Fehler und der augenblicklichen fehlerbezogenen Hazardrate,

$$Z(t, M(t)) = (u_0 - M(t)) z_a(t) = (u_0 - M(t)) \frac{\alpha}{\beta + t}.$$

Die Darstellung einer typischen Realisation der Programmhazardrate im linken Teil von Abb. 26.4 zeigt, dass diese sich anders als bei den bisher vorgestellten Modellen nicht nur zu den Versagens- und Fehlerkorrekturzeitpunkten sprunghaft vermindert, sondern zudem *zwischen* diesen Zeitpunkten kontinuierlich absinkt. Natürlich ist diese Eigenschaft eine Folge der fehlerbezogenen Hazardrate; ihre Interpretation fällt denn auch analog aus: Je länger die Software bereits getestet wurde, desto mehr unterstützt dies die subjektive Einschätzung, dass die Gefährlichkeit der noch nicht behobenen Fehler gering und die Programmqualität somit hoch ist.

Die sich aus der für Binomialmodelle geltenden Gleichung (3) in Verbindung mit der Verteilungsfunktion (4) ergebende Mittelwertfunktion

$$\mu(t) = u_0 F_a(t) = u_0 \left[1 - \left(\frac{\beta}{\beta + t} \right)^{\alpha} \right]$$

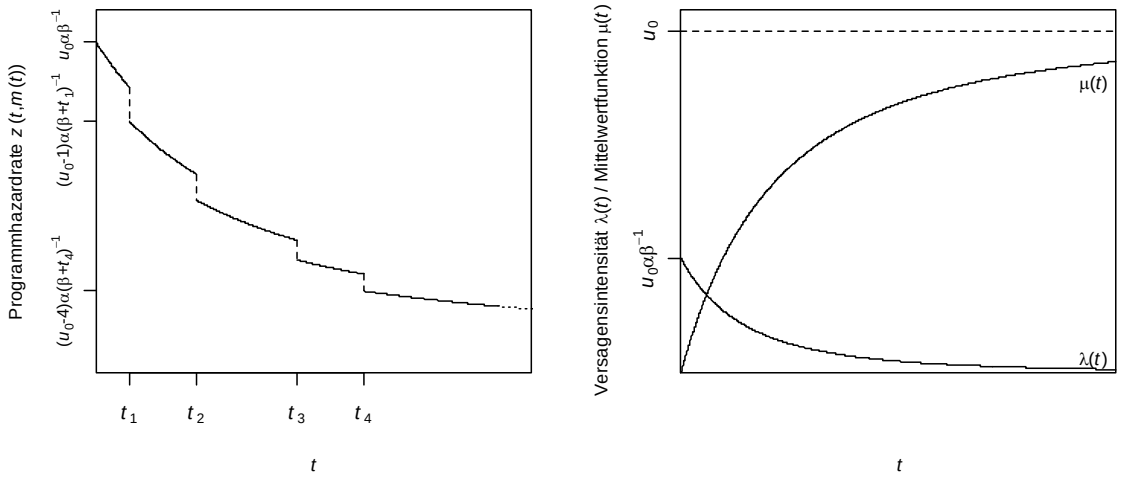

Abb. 26.4: Typische Realisation der Programmhazardrate, Versagensintensität und Mittelwertfunktion gemäß dem Littlewood-Modell

und die mit ihr verbundene Versagensintensität

$$\lambda(t) = u_0 \frac{\alpha}{\beta + t}\left(\frac{\beta}{\beta + t}\right)^{\alpha}$$

sind im rechten Teil von Abb. 26.4 beispielhaft dargestellt.

Waren zum Zeitpunkt t bereits $m(t)$ Versagensfälle aufgetreten, so beträgt die Zuverlässigkeit im Intervall $(t, t+\Delta t]$ dem Littlewood-Modell gemäß

$$R(\Delta t \mid t, m(t)) = \exp\left(-\int_t^{t+\Delta t} z(y, m(t))\, dy\right) = \left(\frac{\beta + t}{\beta + t + \Delta t}\right)^{(u_0 - m(t))\alpha}.$$

Die Anwendung von Gleichung (1) impliziert hierbei die Erwartung, dass bis zu einem weiteren Versagenseintritt die Realisation der Programmhazardrate für alle Zeitpunkte $y > t$ durch $z(y,m(t))$ gegeben ist und somit kontinuierlich absinkt. Diese Annahme ist selbst dann vernünftig, wenn die Prognose am Ende der Testphase erfolgt und die Nutzungsphase betrifft, in der typischerweise kaum Verbesserungen an der Software vorgenommen werden; denn wie gezeigt ist die stetige Verkleinerung der Programmhazardrate in diesem Modell nur subjektiv begründet und nicht durch eine erwartete Fehlerkorrektur bedingt.

Die drei Parameter des Littlewood-Modells (u_0, α und β) können wiederum mit der Maximum-Likelihood-Methode geschätzt werden [33].

26.2.2 Ein Semi-Markovprozess-Modell: Littlewood-Verrall-Modell

Bei den im letzten Abschnitt vorgestellten Modellen hängt die Programmhazardrate jeweils ausschließlich vom aktuellen Zustand $M(t)$ und gegebenenfalls vom aktuellen Zeitpunkt t ab, weshalb sie zu den Markovprozess-Modellen zählen. In einem SZWM aus der Klasse der Semi-Markovprozess-Modelle beeinflusst zusätzlich der letzte

Versagenszeitpunkt $T_{M(t)}$, ein Element der Menge D_t, in einer ganz spezifischen Weise die Programmhazardrate. Diese stellt sich nämlich als Funktion der Zeitspanne *seit* der letzten Versagensbeobachtung dar, der Differenz $t - T_{M(t)}$:

$$Z(t, M(t), D_t) = Z(t - T_{M(t)}, M(t)).$$

T_0 wird hierbei gleich null gesetzt und ist kein Versagenszeitpunkt, sondern der Testbeginn. Der Name der Modellklasse rührt daher, dass zu den Zeitpunkten $T_0, T_1, T_2, \ldots$ der aktuelle Zustand $M(t)$ die einzige Information der Vorgeschichte ist, welche das zukünftige Verhalten des stochastischen Zählprozesses beeinflussen kann, so wie dies in einem Markovprozess-Modell der Fall ist.

Das bedeutendste SZWM aus der Klasse der Semi-Markovprozess-Modelle wurde bereits 1973 von Littlewood und Verrall [34] entwickelt. Die beiden Autoren betrachten die Zufallsvariable $X_j \equiv T_j - T_{j-1}$, die Wartezeit auf das j-te Softwareversagen, gemessen vom $(j-1)$-ten Versagenszeitpunkt (bzw. vom Testbeginn, falls $j-1$ null beträgt und somit die Wartezeit auf den ersten Versagensfall diskutiert wird). Sie gehen davon aus, dass die Hazardrate ϕ_j der Verteilung von X_j die Realisation einer Zufallsvariablen Φ_j darstellt. *Bedingt* auf diese Realisation ist X_j annahmegemäß exponentialverteilt mit Parameter ϕ_j. Für die Zufallsvariable Φ_j unterstellen Littlewood und Verrall, dass sie einer Gammaverteilung mit den Parametern α und $\psi(j)$ folgt, wobei $\psi(j)$ eine monoton steigende Funktion von j ist. Unter diesen Voraussetzungen gilt für jedes beliebige ϕ:

$$P(\Phi_j \le \phi) \ge P(\Phi_{j-1} \le \phi). \tag{6}$$

Dem Littlewood-Verrall-Modell gemäß ist es daher *wahrscheinlich*, dass sich die Softwarequalität nach jeder Fehlerkorrektur verbessert; anders als in den bislang diskutierten Modellen ist dies jedoch nicht *sicher*. Das Modell berücksichtigt also die Möglichkeit, dass die Korrektur eines Fehlers misslingt oder sogar zu weiteren Fehlern in der Software führt, und geht somit auf den zweiten wichtigen Einwand gegen das Jelinski-Moranda-Modell ein. Im Folgenden werden wir diejenige Funktion $\psi(j)$ betrachten, welche in der Diskussion und Anwendung des Littlewood-Verrall-Modells die größte Aufmerksamkeit erfahren hat, $\psi(j) = \beta_0 + \beta_1 j$.

Für die Verteilung der Wartezeit auf das j-te Softwareversagen X_j gilt hier Ähnliches wie im Littlewood-Modell für die Verteilung der Zeitspanne, nach welcher ein bestimmter Fehler ein Versagen verursacht: Während ihre *bedingte* Verteilung annahmegemäß eine Exponentialverteilung ist, folgt für ihre *unbedingte* Verteilung eine Pareto-Verteilung,

$$F_{X_j}(x) = 1 - \left(\frac{\psi(j)}{\psi(j) + x}\right)^{\alpha} = 1 - \left(\frac{\beta_0 + \beta_1 j}{\beta_0 + \beta_1 j + x}\right)^{\alpha},$$

deren Hazardrate

$$\frac{\alpha}{\psi(j) + x} = \frac{\alpha}{\beta_0 + \beta_1 j + x}$$

mit wachsender Wartezeit x absinkt. Die Erklärung dieser Eigenschaft entspricht deshalb auch derjenigen im Littlewood-Modell: Je mehr Zeit bereits seit dem letzten

beobachteten Versagen verstrichen ist, desto weniger gefährlich sind tendenziell die in der Software verbliebenen Fehler.

Die Realisation der Programmhazardrate

$$Z(t - T_{M(t)}, M(t)) = \frac{\alpha}{\psi(M(t)+1) + t - T_{M(t)}} = \frac{\alpha}{\beta_0 + \beta_1 \cdot (M(t)+1) + t - T_{M(t)}} \tag{7}$$

ist aus solchen kontinuierlich fallenden Stücken zusammengesetzt. Der im linken Teil der Abb. 26.5 dargestellte beispielhafte Verlauf lässt allerdings ein Phänomen erkennen, welches in keinem der bislang diskutierten Modelle präsent war: Unter bestimmten Umständen kann die Realisation der Programmhazardrate direkt nach einem Versagensfall höher sein als kurz zuvor. Wie aus Gleichung (7) ersichtlich, ist dies mathematisch gesehen nach dem *j*-ten Versagen genau dann der Fall, wenn die tatsächliche Wartezeit $x_j \equiv t_j - t_{j-1}$ den Wert β_1 überstiegen hat. Das durch das fehlerhafte Verhalten der Software zerstörte Vertrauen in deren Zuverlässigkeit überwiegt dann die durch die Fehlerkorrektur erwartete Verbesserung. Gaudoin [15], S. 62 f., bringt diese Modelleigenschaft auch mit der Möglichkeit zusätzlicher im Rahmen einer Korrekturmaßnahme eingebrachter Fehler in Verbindung, wie sie Gleichung (6) impliziert.

Mittelwertfunktion und Versagensintensität lassen sich für das Littlewood-Verrall-Modell wiederum nicht so einfach herleiten wie für die Binomialmodelle. Musa und andere [47], S. 295 f., ermitteln die Näherungen

$$\mu(t) \approx \frac{1}{\beta_1}\left(\sqrt{\beta_0^2 + 2\alpha\beta_1 t} - \beta_0\right) \quad \text{und} \quad \lambda(t) \approx \frac{\alpha}{\sqrt{\beta_0^2 + 2\alpha\beta_1 t}}.$$

Die approximierte Mittelwertfunktion und ihr typischer Verlauf im rechten Teil von Abb. 26.5 zeigen, dass bei einer unendlich langen Testdauer unendlich viele Versagensfälle zu erwarten sind. Auch deshalb ist das Modell potenziell für Situationen geeignet, in denen bei der Fehlerverbesserung neue Fehler entstehen können.

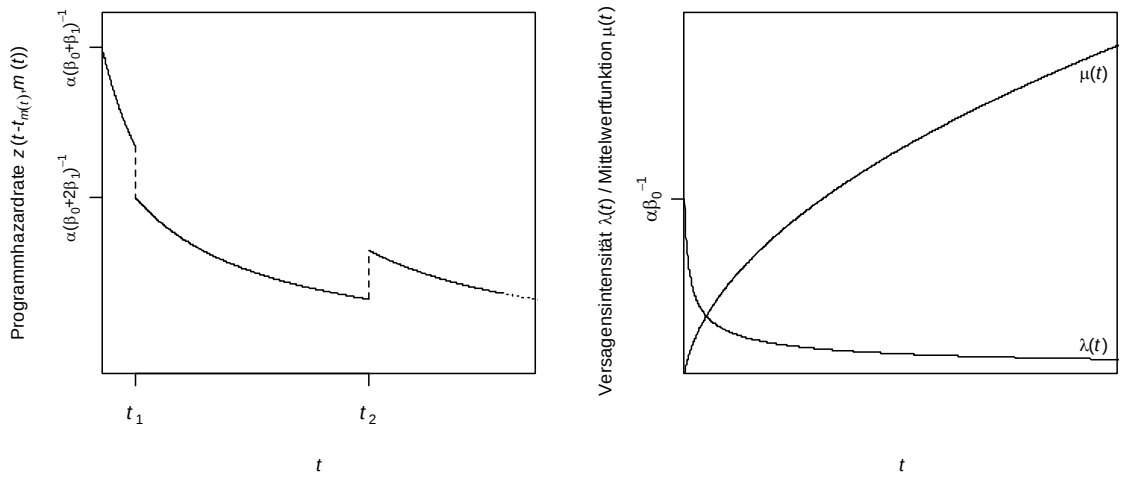

Abb. 26.5: Typische Realisation der Programmhazardrate, approximierte Versagensintensität und approximierte Mittelwertfunktion gemäß dem Littlewood-Verrall-Modell

Wurde die Software bis zum Zeitpunkt t getestet, wobei der letzte der $m(t)$ beobachteten Versagensfälle zum Zeitpunkt $t_{m(t)}$ aufgetreten war, dann beträgt unter Anwendung von Gleichung (1) die Zuverlässigkeit im Intervall $(t, t+\Delta t]$

$$R(\Delta t \mid t, m(t), t_{m(t)}) = \exp\left(-\int_{t}^{t+\Delta t} z(y - t_{m(t)}, m(t))dy\right) = \left(\frac{\beta_0 + \beta_1(m(t)+1) + t - t_{m(t)}}{\beta_0 + \beta_1(m(t)+1) + t + \Delta t - t_{m(t)}}\right)^{\alpha}.$$

Littlewood und Verrall [34] berücksichtigen die Unsicherheit bezüglich des Parameters α, indem sie für ihn eine so genannte uninformative a-priori-Verteilung unterstellen. Die Parameter der Funktion $\psi(j)$ maximieren sie dagegen durch Optimierung einer Metrik, welche die Güte der Anpassung misst. Diesem hybriden Vorgehen stellen Mazzuchi und Soyer [41] ein geschlossenes Bayes-Verfahren gegenüber, welches für jeden der Modellparameter eine a-priori-Verteilung spezifiziert.

26.2.3 Nichthomogene Poissonprozess-Modelle

Im Vergleich zu den in Abschnitt 0 diskutierten Modellen wird die Programmhazardrate bei den SZWM aus der Klasse der Semi-Markovprozess-Modelle von einem *zusätzlichen* Teil der Prozessvorgeschichte beeinflusst. Demgegenüber hängt die Programmhazardrate bei den nun vorgestellten nichthomogenen Poissonprozess-Modellen (kurz: NHPP-Modellen) *überhaupt nicht* von dieser Vorgeschichte ab; sie ist also lediglich eine Funktion der Zeit t:

$$Z(t, M(t), D_t) = z(t).$$

Die Klasse der NHPP-Modelle stellt damit einen Spezialfall der Markovprozess-Modelle dar, bei welchem noch nicht einmal der aktuelle Zustand einen Einfluss auf die zukünftige Entwicklung des Prozesses ausübt.

Dass die Programmhazardrate nach dem Auftreten eines Versagens und der Korrektur des ihn verursachenden Fehlers nicht schlagartig einen anderen Wert annimmt, scheint wenig realistisch zu sein. Eine mögliche Begründung dieser Annahme, die Ascher und Feingold [2], S. 51, als „minimale Reparatur“ bezeichnen, liegt darin, dass ein Computerprogramm sehr viele relativ unbedeutende Fehler enthält; die Bereinigung eines einzelnen Fehlers hat somit kaum eine Auswirkung auf die Programmhazardrate [15], S. 64.

Aber auch die kontinuierliche Veränderung der Programmhazardrate $z(t)$ zwischen den Versagensauftritten muss erklärt werden. Zwei Ansätze kommen in Betracht:

1. Wie im Littlewood- und im Littlewood-Verrall-Modell sind diese Modifikationen ausschließlich subjektiver Natur: Das erhöhte Vertrauen in die Qualität der Software kann die Programmhazardrate zwischen zwei Versagensfällen sinken lassen; die erwartete Zunahme der Fähigkeit der Tester, Fehler aufzuspüren, mag dagegen für ihren Anstieg sorgen.

2. Die Fehlerkorrekturen finden nicht notwendigerweise sofort nach dem beobachteten Softwareversagen, sondern u. U. erst in der Folgezeit statt. Somit kann es auch zwischen den Versagensauftritten zu leichten Änderungen der Programmhazardrate kommen.

Da die Programmhazardrate nicht von der zufälligen Prozessvorgeschichte abhängt, ist sie nicht-stochastisch. (Dies wird in der Notation der Funktion $z(t)$ bereits durch den Kleinbuchstaben z angedeutet.) Ihr Erwartungswert, die Versagensintensität, fällt deshalb mit der Programmhazardrate selbst zusammen:

$$\lambda(t) = E\left[Z(t, M(t), D_t)\right] = E\left[z(t)\right] = z(t)\,. \tag{8}$$

Zur vollständigen Spezifikation eines NHPP-Modells genügt es somit, die Versagensintensität oder ihr Integral, die Mittelwertfunktion $\mu(t)$, anzugeben. Der Name der Modellklasse rührt daher, dass die Anzahl der Versagensfälle im Intervall $(t, t+\Delta t]$ Poisson-verteilt ist mit Erwartungswert $\mu(t+\Delta t) - \mu(t)$. Das Modell ist insofern nicht-homogen, als bei vorgegebener Intervalllänge Δt der Erwartungswert von der Lage des Intervalls abhängt, d. h. von seinem Startzeitpunkt t.

Aus Gleichung (1) in Verbindung mit Gleichung (8) ergibt sich für die Zuverlässigkeit der Software in der Zeitspanne $(t, t+\Delta t]$ allgemein

$$R(\Delta t \mid t) = \exp\left(-\int_t^{t+\Delta t} \lambda(y)\, dy\right) = \exp\left[-\mu(t+\Delta t) + \mu(t)\right]. \tag{9}$$

Dies ist identisch mit der Wahrscheinlichkeit dafür, dass eine Poisson-verteilte Zufallsvariable mit Erwartungswert $\mu(t+\Delta t) - \mu(t)$ den Wert Null annimmt. Gleichung (9) wird generell für die Zuverlässigkeitsprognose auch in der Nutzungsphase eingesetzt. Dies impliziert die Annahme, dass sich die innerhalb der Testphase erwarteten weiteren Veränderungen der Programmhazardrate auch nach dem Software-Release fortschreiben lassen, z. B. weil sie rein subjektiver Natur sind. Yang und Xie [68] stellen diesem Ansatz eine Berechnung der Zuverlässigkeit im Nutzungsbetrieb gegenüber, bei der die Programmhazardrate konstant belassen wird.

Goel-Okumoto-Modell

Das einfachste NHPP-Modell erhält man, wenn man für die Versagensintensität (und damit zugleich für die Programmhazardrate) die Form der Versagensintensität im Jelinski-Moranda-Modell wählt,

$$\lambda(t) = z(t) = \nu\phi\exp\left(-\phi t\right),$$

womit die Mittelwertfunktion

$$\mu(t) = \nu\left[1 - \exp\left(-\phi t\right)\right]$$

beträgt. Der linke Teil von Abb. 26.6 zeigt typische Verläufe der beiden Funktionen.

Anders als im Jelinski-Moranda-Modell ist in diesem von Goel und Okumoto [16] eingeführten SZWM die Anzahl der ursprünglich in der Software enthaltenen Fehler wie auch die Anzahl der bei unendlichem Testaufwand auftretenden Versagensfälle kein fixer Wert u_0. Vielmehr sind beide Größen Poisson-verteilt, und ihr Erwartungswert entspricht dem Parameter ν. Somit kann der Zählprozess $M(t)$ durchaus die Zustände $\nu+1$, $\nu+2$, ... annehmen; es können also mehr als ν Versagensfälle auftreten.

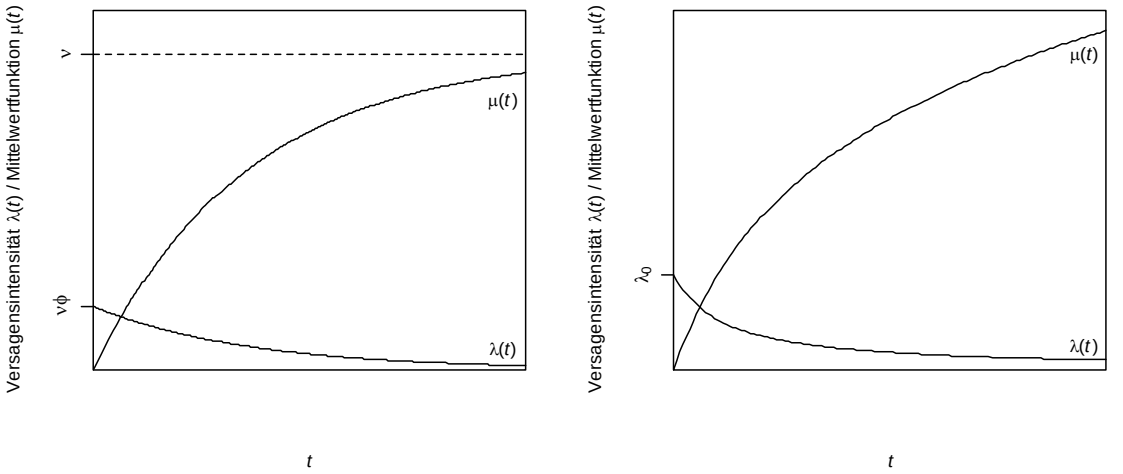

Abb. 26.6: Typische Versagensintensität und Mittelwertfunktion gemäß dem Goel-Okumoto-Modell (links) und dem Musa-Okumoto-Modell (rechts)

Unter der Annahme, dass man die Programmhazardrate des Modells in die Zukunft extrapolieren kann, ergibt sich für die Zuverlässigkeit im Intervall $(t, t+\Delta t]$:

$$R(\Delta t \mid t) = \exp\left[-\mu(t+\Delta t) + \mu(t)\right] = \exp\left[\nu \exp(-\phi(t+\Delta t)) - \nu \exp(-\phi t)\right].$$

Aufgrund der unterschiedlichen Verteilungsannahmen führt die Parameterschätzung mit der Maximum-Likelihood-Methode zu anderen Schätzern als beim Jelinski-Moranda-Modell [16]. Wie für dieses SZWM wurden auch für das Goel-Okumto-Modell Bayes-Ansätze zur Parameterschätzung vorgeschlagen. Da die a-posteriori-Dichte der Modellparameter eine Normierungskonstante aufweist, welche nur schwer berechnet werden kann, stellen Kuo und Yang [29] einen Gibbs-Algorithmus zur Simulation von Zügen aus dieser Dichte vor. Okumura und andere [50] präsentieren dagegen einen weniger rechenintensiven variationellen Bayes-Ansatz, welcher approximative a-posteriori-Dichten in analytisch geschlossener Form liefert. Dieser Ansatz wird in [51] verallgemeinert und verbessert.

Musa-Okumoto-Modell

Ähnlich wie Moranda gehen Musa und Okumoto [46] von einer Versagensintensität aus, die zu Testbeginn schneller abnimmt als in einer späteren Testphase.

Genauer unterstellen sie, dass die Versagensintensität mit der *erwarteten* Anzahl der aufgetretenen Versagensfälle exponentiell absinkt:

$$\lambda(t) = \lambda_0 \exp(-\theta\mu(t)).$$

Berücksichtigt man, dass die Versagensintensität die Ableitung der Mittelwertfunktion ist, so erhält man eine Differenzialgleichung, deren Auflösung zu der Mittelwertfunktion

$$\mu(t) = \frac{1}{\theta}\ln\left(\lambda_0\theta t + 1\right)$$

führt. Wie im Moranda-Modell strebt auch hier die Mittelwertfunktion nicht gegen einen festen Wert. Die Anzahl der bei unendlichem Testaufwand erwarteten Versagensbeobachtungen ist also ebenfalls unendlich. Im rechten Teil von Abb. 26.6 ist der Verlauf dieser Mittelwertfunktion und ihrer Ableitung, der Versagensintensität

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \theta t + 1},$$

beispielhaft dargestellt. Bei Fortschreiben der Versagensintensität folgt für die Zuverlässigkeit im Intervall $(t, t+\Delta t]$:

$$R(\Delta t \mid t) = \exp\left[-\mu(t+\Delta t) + \mu(t)\right] = \left(\frac{\lambda_0 \theta t + 1}{\lambda_0 \theta (t+\Delta t) + 1}\right)^{1/\theta}.$$

Aus den beobachteten Versagensdaten kann man zunächst durch Maximierung der bedingten Likelihood-Funktion (unter der Bedingung, dass bis zum Beobachtungsende t_e insgesamt $m(t_e)$ Versagensfälle aufgetreten sind) einen Schätzer für das Produkt $\lambda_0\theta$ gewinnen und danach den Parameter θ separat schätzen. Aufgrund der Invarianzeigenschaft der Maximum-Likelihood-Schätzung ergibt sich der Schätzer für λ_0 als Quotient dieser beiden Größen. Details finden sich in [46] und in [47], S. 326 und 347.

Goel-Okumoto-Modell mit Weibull-Testaufwand

Wie bereits bemerkt, gehen fast alle SZWM davon aus, dass die Belastung, der ein Programm ausgesetzt ist, im Zeitablauf konstant bleibt. Insbesondere dann, wenn es sich bei dem verwendeten Zeitmaß t um die Kalenderzeit handelt, ist jedoch damit zu rechnen, dass die Intensität der Programmnutzung variiert. Oftmals werden zu Beginn einer Testphase nur wenige Tester eingesetzt (z. B. weil Teile der Software gerade noch programmiert werden), während sich die Anzahl der Tester und damit der Testaufwand erst in der Folgezeit deutlich erhöht, um gegen Ende der Testphase (wenn sich die Anzahl der je Zeiteinheit gefundenen Fehler stark verringert hat) wieder zurückgefahren zu werden. Yamada und andere [67] erweitern das Goel-Okumoto-Modell, indem sie die Verteilung des Testaufwands über die Zeit mit einer Weibull-Funktion beschreiben. Dieser Ansatz führt zu der Mittelwertfunktion

$$\mu(t) = \nu\left[1 - \exp\left(-\phi\alpha\left(1 - \exp(-\beta t^{\gamma})\right)\right)\right]$$

und der mit ihr verbundenen Versagensintensität

$$\lambda(t) = \nu\phi\alpha\beta\gamma t^{\gamma-1} \exp\left(-\phi\alpha\left(1 - \exp(-\beta t^{\gamma})\right) - \beta t^{\gamma}\right),$$

deren typische Verläufe im linken Teil von Abb. 26.7 dargestellt sind. Offensichtlich bewirkt eine zunächst wachsende und dann fallende Testintensität eine im Zeitablauf S-förmige Mittelwertfunktion. Die Weibull-Verteilung ist aber flexibel genug, um auch einen kontinuierlich fallenden Testaufwand modellieren zu können; die Steigung der Mittelwertfunktion nimmt dann stetig ab. Zu beachten ist, dass die Mittelwertfunktion nicht gegen ν konvergiert, sondern gegen einen kleineren Wert. Selbst bei einer unendlichen Testdauer werden also erwartungsgemäß nicht alle Fehler gefunden.

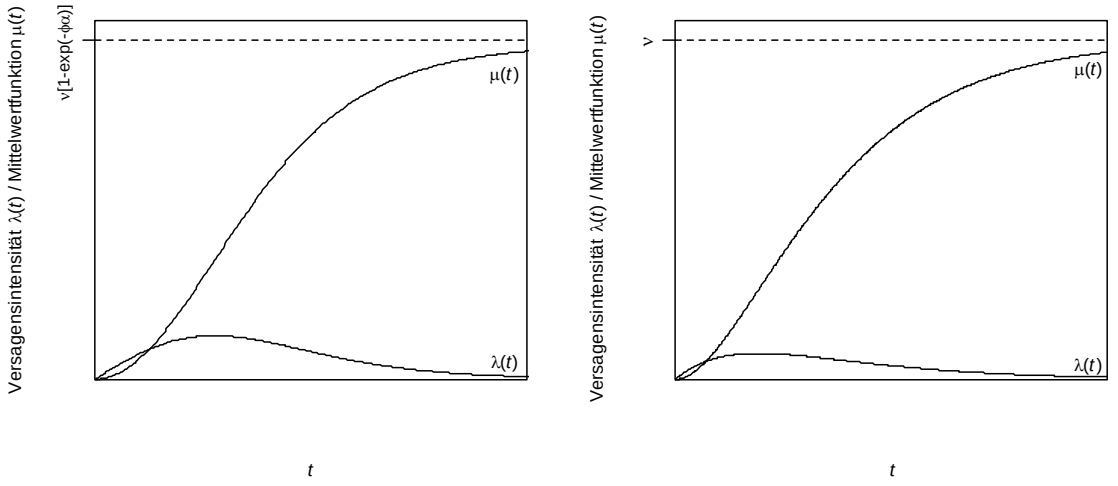

Abb. 26.7: Typische Versagensintensität und Mittelwertfunktion gemäß dem Goel-Okumoto-Modell mit Weibull-Testaufwand (links) und dem verzögert S-förmigen Modell (rechts)

Falls sich die Entwicklung sowohl des Testaufwands als auch des Programmverhaltens in der Zukunft ohne Strukturbruch fortsetzen wird, beträgt die Wahrscheinlichkeit dafür, dass im Intervall $(t, t+\Delta t]$ kein Softwareversagen auftritt,

$$R(\Delta t \mid t) = \exp\left[\nu \exp\left(-\phi\alpha\left(1-\exp(-\beta(t+\Delta t)^{\gamma})\right)\right) - \nu \exp\left(-\phi\alpha\left(1-\exp(-\beta t^{\gamma})\right)\right)\right].$$

Allein auf Versagensbeobachtungen basierend ist die getrennte Schätzung sämtlicher Modellparameter nicht möglich. Die Parameter α und ϕ können nicht identifiziert werden, das Produkt aus ihnen dagegen schon. Yamada und andere [67] gehen davon aus, dass zusätzlich Daten zur Entwicklung des Testaufwands vorliegen, und schlagen ein zweistufiges Schätzverfahren vor.

Verzögert S-förmiges Modell

Schon vor der expliziten Einbeziehung eines variierenden Testaufwands in SZWM waren S-förmige Modelle betrachtet worden. Yamada und andere [66] untersuchen ein NHPP-Modell mit der Mittelwertfunktion

$$\mu(t) = \nu\left[1-(1+\phi t)\exp\left(-\phi t\right)\right] \tag{10}$$

und der aus ihr folgenden Versagensintensität

$$\lambda(t) = \nu\phi^2 t \exp\left(-\phi t\right). \tag{11}$$

Ohba [49] bezeichnet es als „verzögert S-förmiges Modell“, da man mit ihm auch die zeitliche Verzögerung zwischen der Versagensbeobachtung und der Fehlerisolierung (d. h. der Bestätigung der Versagens-Reproduzierbarkeit) modellieren kann: Besteht zwischen der Hazardrate des Prozesses $M(t)$, der die Versagensfälle zählt, und dem momentan erwarteten Fehlergehalt der Software ein proportionales Verhältnis und ist die Hazardrate des Prozesses $G(t)$, welcher die Anzahl der isolierten Fehler zählt,

ihrerseits ein Vielfaches von der erwarteten Anzahl der beobachteten aber noch nicht reproduzierten Versagensfälle, dann hat der Erwartungswert von $G(t)$ die durch Gleichung (10) gegebene Form.

Allerdings wird das Modell durchaus auch auf reine *Versagens*beobachtungen angewandt, wenn diese einen S-förmigen Verlauf aufweisen. Insbesondere erfreut sich das Modell – wie auch andere S-förmige Modelle – in Japan einer großen Beliebtheit zur Anpassung an in Kalenderzeit gemessene Versagensdaten, die aufgrund der im letzten Abschnitt angesprochenen Effekte oftmals eine S-Form besitzen [24].

Kann man davon ausgehen, dass die Versagensintensität auch zukünftig durch Gleichung (11) adäquat beschrieben wird, so beträgt die Zuverlässigkeit im Zeitintervall $(t, t+\Delta t]$

$$R(\Delta t \mid t) = \exp\left[\nu(1+\phi(t+\Delta t))\exp\left(-\phi(t+\Delta t)\right) - \nu(1+\phi t)\exp\left(-\phi t\right)\right].$$

Die Schätzung der beiden Parameter ν und ϕ kann wiederum mit der Maximum-Likelihood-Methode erfolgen [49], [66].

26.2.4 Weitere Ansätze zur Modellvereinheitlichung

Der selbstanregende Punktprozess bildet einen sehr weiten Rahmen, in den sich viele existierende SZWM eingliedern lassen. Er ist aber keineswegs die einzig mögliche Sichtweise zur Vereinheitlichung von Modellen. So haben bereits Langberg und Singpurwalla [31] gezeigt, dass sich sowohl das Goel-Okumoto-Modell als auch das Littlewood-Verrall-Modell gewinnen lässt, indem man das Jelinski-Moranda-Modell in einen Bayes-Kontext einbettet und für seine Parameter spezifische (mitunter degenerierte) a-priori-Verteilungen unterstellt.

Kuo und Yang [29] weisen nach, dass sich für diejenigen NHPP-Modelle, für welche auch bei unendlichem Testaufwand nur eine endliche Anzahl an Versagensfällen ν erwartet wird, die Versagenszeitpunkte als die Ordnungsstatistiken von N unabhängig und identisch verteilten Zufallsvariablen auffassen lassen. Hierbei ist N eine Poisson-verteilte Zufallsvariable mit Erwartungswert ν. Für diejenigen NHPP-Modelle, bei denen die erwartete Zahl der Versagensfälle nicht beschränkt ist, stellen die Versagenszeitpunkte dagegen so genannte Rekorde dar.

Grottke und Trivedi [22], [23] zeigen einen engen Zusammenhang zwischen diesen beiden Arten von NHPP-Modellen auf: In einem NHPP-Modell mit einer endlichen erwarteten Anzahl an Versagensfällen bei unendlicher Testdauer hat jedes X_j (also jede Wartezeit zwischen dem $(j-1)$-ten und dem j-ten Versagensfall) eine so genannte uneigentliche Verteilung, welche eine Wahrscheinlichkeitsmasse bei unendlich aufweist. Es ist demnach möglich, dass es zu keinem j-ten Versagen der Software kommt. Entfernt man all diese Wahrscheinlichkeitsmassen bei unendlich, so ergibt sich ein NHPP-Modell mit einer unbeschränkten erwarteten Zahl der Versagensfälle.

Ein völlig anderer Ansatz zur Modellvereinheitlichung betrachtet nur die Mittelwertfunktionen und führt diese auf verschiedene Einflussfaktoren zurück [20], [21], S. 14 ff. Bei den treibenden Größen, die jeweils miteinander in Beziehung stehen, handelt es sich um die Kalenderzeit, den kumulierten Testaufwand, die Anzahl der ausgeführten

Testfälle und die Codeabdeckung. Eine Differenzialgleichung, welche all diese Faktoren berücksichtigt, enthält als Spezialfälle unter anderem das Jelinski-Moranda-Modell, das Goel-Okumoto-Modell, das Goel-Okumoto-Modell mit Weibull-Testaufwand und das verzögert S-förmige Modell. Der Modellrahmen hilft, die Annahmen eines SZWM den einzelnen Beziehungen zuzuordnen und auf ihre Realitätsnähe zu überprüfen. Zudem kann er als Ausgangspunkt für die Konstruktion neuer Modelle dienen.

26.2.5 Systematisches und nutzungsprofilorientiertes Testen

Alle bislang vorgestellten SZWM gehen implizit davon aus, dass die Software nutzungsprofilorientiert getestet wird. Während des Testens soll ein Programm also in etwa so bedient werden, wie man es von den späteren Nutzern typischerweise erwartet. Grundsätzlich bedeutet dies, dass sich die Gewichtungen der einzelnen Funktionalitäten nach deren (geschätzten) Nutzungsfrequenzen richten, die Eingabewerte aus adäquaten Verteilungen gezogen und die spezifizierten Testfälle in einer zufälligen Reihenfolge ausgeführt werden sollten [45], S. 165 ff.

Weshalb diese Methodik eine wichtige Voraussetzung für die Prognose der Zuverlässigkeit im Nutzungsbetrieb ist, liegt auf der Hand: Falls beim Testen die Software in einer völlig anderen Weise verwendet wird, ändert sich mit ihrer Auslieferung der Prozess, welcher die Versagensfälle generiert. Es ist dann nicht sinnvoll, die Hazardrate des SZWM über das Ende der Testphase hinaus zu extrapolieren. Das nutzungsprofilorientierte Testen kann diesen Strukturbruch verhindern oder zumindest sein Ausmaß verringern.

Allerdings wird diese Teststrategie vielfach für ineffizient und nicht praktikabel gehalten. Der Großteil der softwareproduzierenden Unternehmen setzt so genannte systematische Testtechniken ein. Diese Ansätze versuchen, ausgehend von Informationen über die Funktionalität der Software oder ihre Implementierung Testfälle zu generieren, die möglichst viele unterschiedliche und insbesondere fehleranfällige Bereiche der Software ausführen. (Für eine detailliertere Diskussion des nutzungsprofilorientierten und des systematischen Testens sowie der jeweiligen Vor- und Nachteile siehe [21], S. 6 ff.)

Zwar können ohne genaue Kenntnis der Unterschiede zwischen dem Testprofil und dem Nutzungsprofil die während des systematischen Testens gesammelten Daten nicht zur Prognose der Zuverlässigkeit im Feld verwendet werden. Allerdings ist es auf ihrer Grundlage z. B. möglich, die Anzahl der weiteren Versagensfälle bis zum Testende zu prognostizieren, solange es bis dahin nicht zu Strukturbrüchen kommt. Für den Testmanager und das Programmierteam, welches sich um die Fehlerkorrektur zu kümmern hat, stellen auch diese Informationen wertvolle Planungsgrößen dar.

Die Anwendung eines klassischen SZWM auf Versagensdaten, welche dem systematischen Testen entstammen, führt jedoch nicht unbedingt zu vertrauenswürdigen Ergebnissen. Der Grund hierfür liegt darin, dass die Modelle für das nutzungsprofilorientierte Testen geschaffen wurden und sich dies mitunter in den Strukturen der unterstellten Programmhazardrate und der von ihr abgeleiteten Größen widerspiegelt. So ergibt sich z. B. die Mittelwertfunktion des Jelinski-Moranda- und des Goel-Okumoto-Modells, wenn man ein Ziehen von Codekonstrukten mit Zurücklegen unterstellt [53]; dieser

Aufbau ähnelt stark dem nutzungsprofilorientierten Testen mit einem homogenen Nutzungsprofil. Ausgehend von dem im letzten Abschnitt erwähnten Modellrahmen, der aus sukzessiven Beziehungen zwischen treibenden Faktoren besteht, wird in [21], S. 37 ff., ein Modell für die Entwicklung der Anzahl der Versagensfälle während des systematischen Testens hergeleitet.

26.2.6 Evaluierung und Verbesserung der Modellgüte

Aufgrund der Vielzahl der existierenden SZWM scheint für jeden Fall ein adäquates Modell bereitzustehen. Das übergroße Angebot hat aber auch Nachteile, ist es doch Ausdruck der Tatsache, dass keines der SZWM für jeden Datensatz gute Ergebnisse liefert. Schlimmer noch: Da jedes Modell nur einen kleinen Teil der mannigfaltigen technischen und sozialen Einflussfaktoren des versagensverursachenden Prozesses abbilden kann, ist es nicht möglich, im Vorfeld der Datenerhebung mit Sicherheit zu entscheiden, welches der Modelle am besten zu den Versagensbeobachtungen passen wird [7].

Umso wichtiger ist es, für einen vorhandenen Datensatz die Güte verschiedener SZWM zu vergleichen. Hierbei gibt es eine Reihe von Kriterien, die unterschiedliche Aspekte der Modellqualität operationalisieren und erfassen. Das grundsätzliche Vorgehen zur Berechnung dieser Maße ist dabei immer gleich: Beginnend mit den ersten z. B. fünf Datenpunkten des gesamten Datensatzes werden die Parameter eines SZWM geschätzt und zur Prognose einer bestimmten Größe (z. B. des nächsten Versagenszeitpunkts) bzw. deren Verteilung verwendet. Unter Hinzunahme jeweils eines weiteren Datenpunkts zu dem gestutzten Datensatz wird diese Prozedur sukzessive wiederholt. Man simuliert also die begleitende Anwendung des Modells während des gesamten bisherigen Projektverlaufs. Aus dem Vergleich der einzelnen Prognosen untereinander bzw. mit den tatsächlichen Beobachtungen errechnet sich schließlich das Gütekriterium für das jeweilige SZWM.

Folgende konkrete Maße werden oftmals betrachtet:

1. Absolute relative Prognosefehler [12], S. 3 f., [21], S. 78 f.:

   Zur Beurteilung der Qualität der *kurzfristigen* Prognose wird in jedem Schritt die geschätzte Anzahl der Versagensfälle zum nächsten Versagenszeitpunkt mit dem tatsächlichen Wert verglichen und der Absolutbetrag der relativen Abweichung bestimmt. Der kurzfristige absolute relative Prognosefehler ergibt sich dann als Mittelwert all dieser Größen. Um das *langfristige* Verhalten eines Modells quantifizieren zu können, ist der maximale Prognosehorizont zu wählen, für den eine Gegenüberstellung mit der Realität möglich ist. Deshalb wird für den so genannten mittleren absoluten relativen Prognosefehler für jeden gestutzten Datensatz die prognostizierte Anzahl an Versagensfällen bis zum Ende des Beobachtungszeitraums mit dem tatsächlichen Wert verglichen. Selbstredend ist ein SZWM umso besser, je geringer seine Prognosefehler ausfallen.

2. Variabilitätsmaße [1], [21], S. 80 f.:

   Um als Planungsgrundlage dienen zu können, dürfen sich die von einem Modell gelieferten Qualitätseinschätzungen bei der Hinzunahme einer weiteren Beobachtung nicht zu stark verändern. Zur Beurteilung des Ausmaßes dieser unerwünschten Variabilität berechnet man aus der Sequenz der Prognosen (z. B. der Zeitspanne bis zum nächsten Softwareversagen) für je zwei aufeinanderfolgende Werte den Absolutbetrag der relativen Abweichung. Das Variabilitätsmaß ergibt sich dann als Summe dieser Abweichungsgrößen; je kleiner sein Wert ist, desto besser.

3. Präquenzieller Likelihood-Wert [1]:

   Mit der Schätzung eines (zeitbasierten) SZWM aufgrund der bisherigen Beobachtungen wird indirekt zugleich die Verteilung der Zeit bis zum nächsten Versagen prognostiziert. Falls das Modell adäquat ist, sollte man erwarten, dass die später eintretende Realisation aus einem Bereich dieser Verteilung stammt, welcher eine große Eintrittswahrscheinlichkeit aufweist. Bei einer stetigen Zufallsvariablen sollte die Dichtefunktion an der Stelle der Beobachtung tendenziell einen hohen Wert annehmen. Evaluiert man in der sequenziellen Modellanwendung jede der Prognosedichten an der jeweiligen Realisation und multipliziert die so erhaltenen Größen, dann ergibt sich ein Maß für die Plausibilität des Modells anhand des gesamten Datensatzes, welches als präquenzieller Likelihood-Wert (*Prequential Likelihood*) bezeichnet wird. Zum Vergleich zweier SZWM bildet man den Quotienten ihrer beiden präquenziellen Likelihood-Werte. Tendiert dieses präquenzielle Likelihood-Verhältnis mit zunehmender Datensatzlänge gegen unendlich, dann ist das Modell, dessen präquenzieller Likelihood-Wert im Zähler steht, dem anderen vorzuziehen; tendiert es gegen null, so trifft das Gegenteil zu.

4. u-Plot und y-Plot [1], [7]:

   Obwohl die tatsächlich beobachteten Wartezeiten bis zum jeweils nächsten Versagensfall überwiegend aus denjenigen Bereichen der Prognosedichten stammen sollten, welche eine hohe Wahrscheinlichkeitsmasse umfassen (wie vom präquenziellen Likelihood-Wert betont), sind durchaus auch – einige wenige – Realisationen aus den Rändern der Verteilungen zu erwarten. So dürften z. B. etwa 5% der Werte kleiner als diejenigen Schranken sein, die mit fünfprozentiger Wahrscheinlichkeit unterschritten werden. Beim so genannten u-Plot handelt sich um ein grafisches Instrument, mit dessen Hilfe überprüft werden kann, ob die Beobachtungen in diesem Sinne zu der Gestalt der von einem Modell prognostizierten Verteilungsfunktionen passen. Hierbei kann nicht nur die Stärke der Abweichung quantifiziert und mit derjenigen eines anderen SZWM verglichen werden. Es zeigt sich zudem, ob die tatsächlichen Wartezeiten bis zum nächsten Softwareversagen tendenziell in den oberen (unteren) Rändern der prognostizierten Verteilungen liegen und das Modell somit die Zuverlässigkeit der Software systematisch unterschätzt (überschätzt). Falls allerdings für eine Hälfte der Daten die Prognosen zu optimistisch sind, während sie für die andere Hälfte zu pessimistisch ausfallen, können sich diese gegensätzlichen Abweichungen des Modells von der Wirklichkeit derart ausgleichen, dass sie im u-Plot nicht zu entdecken sind. Der auf den Werten des u-Plots aufbauende y-Plot kann solche Trends identifizieren.

Hat man mithilfe eines Gütemaßes erkannt, dass ein SZWM bei der Prognose systematische Fehler macht, ist es möglich, dieses Wissen zur Verbesserung der Vorhersagen zu nutzen. Dies ähnelt dem Vorgehen eines Schützen, der bei seinen bisherigen Schüssen immer links am Ziel vorbeigeschossen hat: Die Lage rekapitulierend wird er beim nächstem Mal weiter nach rechts zielen. Brocklehurst und andere [8] schlagen eine solche „Rekalibrierung" von Prognosen vor, die auf den Ergebnissen des (geglätteten) u-Plots basiert. Sie zeigen, dass diese Technik die Vorhersagen verschiedener SZWM aneinander angleicht und dass zudem dem präquenziellen Likelihood-Verhältnis gemäß die rekalibrierten „Modelle" ihren ursprünglichen Varianten vorzuziehen sind.

Einen deutlich einfacheren Ansatz wählen Lyu und Nikora [37]. Sie raten dazu, eine Zuverlässigkeitsprognose als arithmetisches Mittel der Vorhersagen mehrerer SZWM zu berechnen. Insbesondere empfehlen sie die Mittelung der Prognosen des Goel-Okumoto-Modells (welches generell als zu optimistisch gilt), des Littlewood-Verrall-Modells (welches zu pessimistischen Vorhersagen tendiert) und des Musa-Okumoto-Modells (dessen Verzerrungsrichtung stärker variiert). Bei Erweiterungen der Methodik können die Gewichte für die einzelnen SZWM unterschiedlich ausfallen und sich sogar dynamisch nach der relativen Güte der Anpassung des jeweiligen Modells an die Daten richten [38]. In dieser adaptiven Variante handelt es sich bei den Gewichten um so genannte Bayes-Faktoren; diese sind eng mit den präquenziellen Likelihood-Werten der verschiedenen Modelle verbunden [58], S. 148 ff.

Ein grundsätzlicher Nachteil der Rekalibrierung und der Mittelung von Prognosen liegt darin, dass die Annahmen der ursprünglichen Modelle und die Interpretierbarkeit einzelner Modellparameter (z. B. des Parameters u_0 als der Anzahl der zu Beginn vorhandenen Softwarefehler im Jelinski-Moranda-Modell) verloren gehen. Die Gesamtheit aus Modell(en), Schätz- und Prognoseverfahren wird vollends zur Blackbox.

26.3 Weitere Modellklassen

Zwar haben die SZWM in der Literatur zur Schätzung und Prognose der Zuverlässigkeit einer Software die größte Aufmerksamkeit erfahren. Es handelt sich bei ihnen aber keineswegs um die einzige existierende Modellklasse. Der Vollständigkeit halber sollen in diesem Abschnitt einige weitere Modellansätze skizziert werden. Völlig ausklammern wollen wir aus Platzgründen Verfahren zum Nachweis eines geforderten Zuverlässigkeitsniveaus im Rahmen von Akzeptanztests (siehe [47], S. 201 ff.), welche auf der statistischen Theorie des sequenziellen Testens [62] beruhen.

26.3.1 Stichprobenmodelle

Anders als die in Abschnitt 0 beschriebenen SZWM versuchen die in diesem Abschnitt vorgestellten Modelle nicht, die Entwicklung der Zuverlässigkeit oder der Anzahl der Versagensfälle während einer Testphase mit Fehlerkorrektur nachzuvollziehen und vorherzusagen. Vielmehr dienen sie dazu, den aktuellen Fehlergehalt oder die Zuverlässigkeit einer Software zu bestimmen. Insofern mag man sie eher als Schätz- denn als Prognosemodelle bezeichnen. Allerdings ist zu beachten, dass Zuverlässigkeitswerte als Wahrscheinlichkeiten für einen Versagenseintritt bei zukünftiger Nutzung immer auch Vorhersagen sind, selbst wenn das Softwareprodukt unverändert bleibt.

Zu ihrer Schätzung benötigen die Modelle keine Informationen über die Entwicklung der Anzahl der Versagensfälle im Zeitablauf. Die Daten der sukzessive durchgeführten Tests können gruppiert vorliegen, entweder in Form einer globalen Stichprobe oder in zwei Stichproben getrennt.

Nelson-Modell

Dieses Modell [61], S. 217 ff., gründet sich auf derjenigen Zuverlässigkeitsdefinition, welche die Länge der „Nutzungsperiode“ anhand der Anzahl der Programmläufe misst (siehe Abschnitt 0). Genauer wird hier die Zuverlässigkeit R als Wahrscheinlichkeit dafür verstanden, dass im Rahmen *eines* Laufs kein Versagen auftritt. Ein Programmlauf ist dabei die Ausführung der Software mit einer bestimmten Kombination von Eingabewerten für die Inputvariablen. Diese entstammt der sehr großen, aber endlichen Menge aller möglichen Wertekombinationen. Kam es während des Testens bei m von insgesamt n Programmläufen zu einem Softwareversagen, dann lautet die Zuverlässigkeitsschätzung nach dem Nelson-Modell

$$\hat{R} = 1 - \frac{m}{n}.$$

Damit dieser Wert auch ein unverzerrter Schätzer für die versagensfreie Programmausführung im normalen Nutzungsbetrieb sein kann, müssen natürlich die Auswahlwahrscheinlichkeiten der Inputkombinationen denjenigen entsprechen, welche nach dem Release-Zeitpunkt vorherrschen werden; kurz: Es muss nutzungsprofilorientiert getestet werden. Obwohl der Schätzer unter dieser Voraussetzung unverzerrt ist [61], S. 222 ff., benötigt man eine große Anzahl an Testläufen, um seine Varianz gering zu halten und somit ein hohes Vertrauen in die Punktschätzung legen zu können [3].

Dass das Modell an das nutzungsprofilorientierte Testen gebunden ist und deshalb nicht während des Testens gemäß systematischer Strategien verwendet werden kann, wird als weiterer Nachteil gesehen [3].

Brown-Lipow-Modell

Eine Lösung des letztgenannten Problems bietet der Ansatz von Brown und Lipow [9]. Zu seiner Anwendung ist es nicht nötig, dass dem Nutzungsprofil gemäß getestet wird; dieses Profil muss aber in folgender Weise explizit spezifiziert sein: Die große Menge der möglichen Eingabekombinationen sei in Teilmengen $Z_1, Z_2, ..., Z_K$ partitioniert. Bei diesen Teilmengen kann es sich beispielsweise um Äquivalenzklassen handeln, deren Elemente bei ihrer Eingabe erwartungsgemäß jeweils die gleiche Reaktion der Software bewirken. Das Nutzungsprofil muss dann in Form der Auftrittswahrscheinlichkeiten all dieser Teilmengen bei normaler Programmnutzung, $P(Z_1), P(Z_2), ..., P(Z_K)$, bekannt sein. Als Ergebnis der Testdurchführung ist für jedes Z_j zum einen die Anzahl n_j der Programmläufe festzuhalten, deren Eingabekombinationen zu dieser Teilmenge gehören. Des Weiteren muss jeweils gezählt werden, wie viele der n_j Läufe zu einem Versagen führen; für Z_j sei dieser Wert mit m_j bezeichnet. Die geschätzte augenblickliche Zuverlässigkeit der Software bei Bedienung dem Nutzungsprofil entsprechend beträgt dann

$$\hat{R} = 1 - \sum_{j=1}^{K} \frac{m_j}{n_j} P(Z_j) .$$

Nelson [48] überträgt diese Formel auf eine Situation, in der die Läufe aller spezifizierten Testfälle zu *keinem* Softwareversagen führen (z. B. weil bereits zuvor alle Testfälle durchgeführt und die durch sie aufgedeckten Fehler bereinigt wurden). Die Zuverlässigkeitsschätzung errechnet er als

$$\hat{R} = 1 - \sum_{j=1}^{K} \varepsilon_j P(Z_j) ,$$

wobei ε_j die Wahrscheinlichkeit dafür bezeichnet, dass eine beliebige aus Z_j gewählte Eingabekombination zu einem Versagen führt. Für die Bestimmung dieser ε_j-Werte gibt Nelson heuristische Regeln an, welche unter anderem die Anzahl der Testfälle berücksichtigen, die sich auf die Teilmenge Z_j beziehen.

Mills-Modell

Ist man nicht an der Zuverlässigkeit, sondern lediglich an der Anzahl der Programmfehler interessiert, so kann man sich so genannte Capture-Recapture-Modelle zunutze machen, statistische Modelle, welche ursprünglich zur Schätzung der Größe von Populationen (z. B. der Anzahl der Fische in einem Teich) verwendet wurden [28], S. 248 ff. In einer spezifischen Form [52], S. 159 f., wurden sie erstmals von Mills auf das Gebiet des Softwaretestens übertragen. In ein Programm, welches eine unbekannte Anzahl von Fehlern u_0 aufweist, werden bewusst u_1 Fehler eingebaut. Es sei angenommen, dass alle Fehler in etwa die gleiche Entdeckungswahrscheinlichkeit besitzen und insbesondere die u_1 „gesäten“ Fehler nicht leichter oder schwerer zu finden sind als die u_0 von Anfang an vorhandenen. Zudem liege keine Interaktion zwischen den verschiedenen Fehlern vor.

Wird nun eine Reihe von Testfällen durchgeführt, wobei insgesamt f Fehler gefunden werden, dann ist die Anzahl derjenigen unter ihnen, bei denen es sich um gesäte Fehler handelt, zufällig. Unter den oben genannten Voraussetzungen folgt diese Zufallsvariable, die mit F_1 bezeichnet sei, einer hypergeometrischen Verteilung. Die Wahrscheinlichkeit dafür, dass sich genau f_1 gesäte Fehler unter den f entdeckten befinden, beträgt also

$$P(F_1 = f_1; u_0, u_1, f) = \frac{\binom{u_0}{f - f_1}\binom{u_1}{f_1}}{\binom{u_0 + u_1}{f}} .$$

Wurden tatsächlich f_1 gesäte Fehler wiedergefunden, so ergibt sich mittels der Maximum-Likelihood-Methode folgende Schätzung für den Parameter u_0, die Anzahl der ursprünglichen Fehler [10], S. 108:

$$\hat{u}_0 = \left\lfloor \frac{u_1 (f - f_1)}{f_1} \right\rfloor .$$

Hierbei bezeichnet $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner oder gleich x ist. Der Schätzwert für u_0 entspricht also in etwa derjenigen Größe, die man erhält, wenn man die Anzahl der gefundenen ursprünglichen Fehler ($f - f_1$) durch die Entdeckungsquote bei den gesäten Fehlern (f_1 / u_1) dividiert. Offensichtlich wird dieser Ansatz u_0 tendenziell unterschätzen, wenn die bewusst eingefügten Fehler leichter zu finden sind als die ursprünglichen und damit erwartungsgemäß über eine höhere Entdeckungsquote verfügen.

Basin-Modell

Ein etwas veränderter Aufbau des Experiments, der kein Einbringen weiterer Fehler erfordert, wird mit Basin in Verbindung gebracht [10], S. 113. Das Fangen und Wiederfangen, welches in dem Begriff „Capture-Recapture-Modell“ zum Ausdruck kommt, wird hier in der Form des unabhängigen Testens der Software durch zwei Personen realisiert. Von den insgesamt u_0 Fehlern habe der erste Tester f_1 und der zweite Tester f_2 entdeckt. Falls die Schwierigkeit des Auffindens für jeden Fehler gleich groß ist und zudem nicht von der Person des Testers abhängt, beträgt die Wahrscheinlichkeit dafür, dass genau w der f_2 vom zweiten Tester aufgespürten Fehler bereits von seinem Kollegen entdeckt worden waren,

$$P(W = w; u_0, f_1, f_2) = \frac{\binom{f_1}{w}\binom{u_0 - f_1}{f_2 - w}}{\binom{u_0}{f_2}}.$$

Wiederum tritt also die hypergeometrische Verteilung in Erscheinung.

Unter Verwendung des für w tatsächlich beobachteten Werts lautet der Maximum-Likelihood-Schätzer für die Gesamtzahl an Fehlern in der Software

$$\hat{u}_0 = \left\lfloor \frac{f_1 f_2}{w} \right\rfloor.$$

Wurden vom zweiten Tester also $(w / f_1) = y\%$ der zuvor vom ersten Tester gefundenen Fehler wiederentdeckt, so kann man davon ausgehen, dass die f_2 Fehler etwa $y\%$ der Gesamtfehlerzahl ausmachen.

Auf Probleme der Verwendung von Capture-Recapture-Modellen zur Schätzung des Fehlergehalts einer Software weist Isoda [26] hin.

26.3.2 Modelle zur Prognose von Softwarefehlern

Die bisher behandelten Modelle haben gemein, dass sie Daten aus der Ausführung des betrachteten Programms verwenden, um Rückschlüsse über dessen Qualität zu ziehen. In diesem Abschnitt sind einige Ansätze zusammengefasst, die aufgrund anderer (typischerweise bereits vor der Testphase verfügbarer) Informationen versuchen, den Fehlergehalt der Software vorherzusagen. Gegenüber den anderen Modellen unterscheiden sich diese Querschnittsmodelle grundlegend darin, dass zur Schätzung ihrer Parameter nicht nur die Daten eines einzigen Projekts verwendet werden. Viel-

mehr greift man entweder auf in der Literatur publizierte Erfahrungs- und Schätzwerte zurück, oder man schätzt die Parameter basierend auf einer Sammlung von früheren Projekten des eigenen Unternehmens. In jedem Fall unterstellt man, dass die Zusammenhänge, die anhand der für die Modellspezifikation genutzten Projekte identifiziert wurden, auch für die zukünftigen Projekte gültig sind. Die Prognoseergebnisse sind mit umso größerer Vorsicht zu genießen, je stärkere Zweifel an der Vergleichbarkeit der Projekte bestehen.

Multiplikative Modelle

Um multiplikative Modelle handelt es sich z. B. bei dem Modell des Rome Laboratory der Air Force (siehe [13] und [30], S. 7-4 ff.) sowie bei dem Ansatz von Malaiya und Denton [39]. In ihnen ergibt sich die prognostizierte Fehlerdichte – die Anzahl der Fehler je 1000 Sourcecode-Zeilen – als Produkt einer Reihe von Faktoren, deren Werte in Abhängigkeit von den Gegebenheiten der Software und des gesamten Entwicklungsprojekts bestimmt werden. Beide Modelle verfügen über einen Faktor, welcher eine Basis-Fehlerdichte repräsentiert. Im Modell des Rome Laboratory wird der Wert dieses Faktors anhand einer Checkliste ermittelt, welche die Schwierigkeit der Entwicklung der vorliegenden Art von Software beurteilt; bei Malaiya und Denton beruht er auf der von dem betrachteten Unternehmen im Durchschnitt erreichten Fehlerdichte. Die weiteren Faktoren berücksichtigen Aspekte der verwendeten Entwicklungsmethoden, der institutionalisierten Entwicklungs- und Testprozesse, der Eignung der Mitarbeiter und der Struktur des implementierten Codes. All diese Faktoren weisen einen Wertebereich um die Zahl Eins auf. Je nach Ausprägung der einzelnen Aspekte wird also die Basis-Fehlerdichte erhöht oder verringert. So nimmt z. B. der Programmierteam-Faktor in Malaiyas und Dentons Modell bei einer durchschnittlichen Leistungsfähigkeit den Wert Eins an und bei einem hohen (niedrigen) Leistungsniveau den Wert 0,4 (2,5). Der Vorteil dieses multiplikativen Aufbaus besteht darin, dass bei Fehlen einzelner Informationen die jeweiligen Faktoren weggelassen (und dabei faktisch auf ihren Grundwert Eins gesetzt) werden können. Da es möglich ist, die Faktoren des Rome-Laboratory-Modells den Entwicklungsphasen Analyse, Design und Implementierung/Test zuzuordnen, kann man somit für jeden Entwicklungsstand ein Submodell aufstellen, welches eine Teilmenge der Faktoren umfasst [30], S. 7-4 ff.

Lineare Regressionsmodelle

Lineare Regressionsmodelle versuchen, eine abhängige Variable y auf eine Linearkombination von erklärenden Variablen $x_1, ..., x_k$ zurückzuführen:

$$y = \alpha_0 + \alpha_1 x_1 + ... + \alpha_k x_k + \eta \,.$$

Hierbei steht η für die zufällige Abweichung von dem linearen Zusammenhang, für welche unter anderem ein Erwartungswert von null unterstellt wird.

In der konkreten Anwendung der Fehlerprognose handelt es sich bei der abhängigen Variablen um die Anzahl der Fehler im Programmcode oder um die Fehlerdichte. Mitunter wird jedoch auch auf den natürlichen Logarithmus der Fehlerdichte zurückge-

griffen; ein Vorteil dieses Vorgehens liegt darin, dass der Logarithmus im Gegensatz zur Fehlerdichte selbst nicht auf Werte größer oder gleich null beschränkt ist.

Takahashi und Kamayachi [60] definieren neun quantitative Indikatoren, die potenziell einen Einfluss auf die Anzahl der Programmfehler haben. Bei denjenigen drei Variablen, die für ihren Datensatz aus 30 Projekten die größte Erklärungskraft für die Gesamtfehlerzahl zeigen, handelt es sich um die Häufigkeit von Änderungen der Programmspezifikation (gemessen in Seiten der Änderungswünsche), die durchschnittliche Programmiererfahrung der Entwickler (in Jahren) und den Umfang der Designdokumente (in Seiten). Für diese Variablen stellen die Autoren ein lineares Regressionsmodell mit der Anzahl der Programmfehler als exogene Variable auf.

Zhang und Pham [69] erweitern Takahashis und Kamayachis Liste der Einflussfaktoren deutlich. Mit einem Fragebogen erheben sie bei verschiedenen Gruppen von Mitarbeitern in Softwareunternehmen (z. B. Managern, Programmierern und Testern) die subjektiv empfundene Bedeutung dieser Faktoren für die Zuverlässigkeit der entwickelten Software, ohne allerdings anhand echter Projektdaten die Erwartungen zu verifizieren oder ein Regressionsmodell zu schätzen.

In [21] wird eine Auswahl der von Zhang und Pham zusammengetragenen Einflussfaktoren weiter operationalisiert und damit objektiv messbar gemacht. Zudem enthält ein ausführlicher Fragebogen [21], S. 223 ff., Fragen und detaillierte Szenarien, mithilfe derer die Reife von Softwareentwicklungsprozessen in Anlehnung an den im Standard ISO/IEC 15504 [25] definierten SPICE-Framework bestimmt werden kann. Die Analyse der 13 verfügbaren Projektdatensätze führt zu einem linearen Regressionsmodell, bei dem die Fehlerdichte durch eine selektive Reifegradbewertung, das Verhältnis zwischen der tatsächlichen und der geplanten Entwicklungsdauer und den Anteil der nach der Spezifikationsphase geänderten Anforderungen erklärt wird.

Insbesondere dann, wenn es sich bei den exogenen Faktoren um Maße der Programmkomplexität handelt, welche erwartungsgemäß stark miteinander verbunden sind, können sich die geschätzten Regressionskoeffizienten bei der Aufnahme weiterer erklärender Variablen deutlich verändern. Um dies und weitere Probleme der so genannten Multikollinearität [19], S. 59 ff., in den Griff zu bekommen, schlagen Munson und Khoshgoftaar [44] die Anwendung der Hauptkomponentenanalyse zur Gewinnung von orthogonalen (unabhängigen) Faktoren vor.

Eine Übersicht über weitere – nicht notwendigerweise lineare – Regressionsmodelle für Fehlerdaten findet sich bei Cai [10], S. 47 ff.

26.4 Abschließende Bemerkung

Dieses Kapitel gibt einen knappen Überblick über verschiedene Ansätze zur Prognose von Softwarezuverlässigkeit, Softwareversagensfällen und Softwarefehlern. Da im Rahmen eines Softwareentwicklungsprojekts das korrekte Softwareverhalten zumeist nur eines der zu beachtenden Kriterien darstellt (neben dem Funktionsumfang, der Entwicklungszeit, den Lebenszykluskosten, usw.), kann es sinnvoll sein, die hier diskutierten Modelle als Elemente umfassenderer Optimierungsprobleme einzusetzen. So betrachten z. B. Pham [52], S. 315 ff., und Yamada [65] Ansätze zur Bestimmung der-

jenigen Testdauer, welche die Gesamtkosten der Testdurchführung und der Gewährleistung minimiert.

Literaturverzeichnis

[1] Abdel-Ghaly, A.A., Chan, P.Y. und Littlewood, B., Evaluation of competing software reliability predictions, IEEE Transactions on Software Engineering 12 (1986), S. 950 ff.

[2] Ascher, H. und Feingold, H., Repairable systems reliability – Modeling, inference, misconceptions and their causes, New York 1984.

[3] Bastani, F.B. und Ramamoorthy, C.V., Software reliability, in: Krishnaiah, P.R. und Rao, C.R. (Hrsg.), Handbook of statistics, Vol. 7, Amsterdam 1988, S. 7 ff.

[4] Bauer, E., Zhang, X. und Kimber, D.A., Practical system reliability, Hoboken, 2009.

[5] Belli, F., Grochtmann, M. und Jack, O., Erprobte Modelle zur Quantifizierung der Software-Zuverlässigkeit, Informatik Spektrum 21 (1998), S. 131 ff.

[6] Boland, P.J. und Singh, H., A birth-process approach to Moranda's geometric software-reliability model, IEEE Transactions on Reliability 52 (2003), S. 168 ff.

[7] Brocklehurst, S. und Littlewood, B., Techniques for prediction analysis and recalibration, in: Lyu, M.R. (Hrsg.), Handbook of software reliability engineering, New York 1996, S. 119 ff.

[8] Brocklehurst, S., Chan, P.Y. und Littlewood, B., Recalibrating software reliability models, IEEE Transactions on Software Engineering 16 (1990), S. 458 ff.

[9] Brown, J.R. und Lipow M., Testing for software reliability, Proceedings of the International Conference on Reliable Software, New York 1975, S. 518 ff.

[10] Cai, K.-Y., Software defect and operational profile modeling, Boston 1998.

[11] Chen, Y. und Singpurwalla, N.D., Unification of software reliability models by self-exciting point processes, Advances in Applied Probability 29 (1997), S. 337 ff.

[12] Denton, J.A., Accurate software reliability estimation, Master's thesis, Colorado State University, Fort Collins 1999.

[13] Farr, W., Software reliability modeling survey, in: Lyu, M.R. (Hrsg.), Handbook of software reliability engineering, New York 1996, S. 71 ff.

[14] Forman, E.H. und Singpurwalla, N.D., An empirical stopping rule for debugging and testing computer software, Journal of the American Statistical Association 72 (1997), S. 750 ff.

[15] Gaudoin, O., Outils statistiques pour l'évaluation de la fiabilité des logiciels, Thèse de doctorat, Université de Joseph Fourier – Grenoble 1, Grenoble 1990.

[16] Goel, A.L. und Okumoto, K., Time-dependent error-detection model for software reliability and other performance measures, IEEE Transactions on Reliability 28 (1979), S. 206 ff.

[17] Gokhale, S.S., Software reliability, in: Wah, B.W. (Hrsg.), Wiley encyclopedia of computer science and engineering, Hoboken 2009, S. 2669 ff.

[18] Gokhale, S.S., Marinos, P.N. und Trivedi, K.S., Important milestones in software reliability modeling, Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering, Skokie 1996, S. 345 ff.

[19] Greene, W.H., Econometric analysis, 6. Auflage, Upper Saddle River 2007.

[20] Grottke, M., A vector Markov model for structural coverage growth and the number of failure occurrences, Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering, Los Alamitos 2002, S. 304 ff.

[21] Grottke, M., Modeling software failures during systematic testing – The influence of environmental factors, Aachen 2003.

[22] Grottke, M. und Trivedi, K.S., On a method for mending time to failure distributions, Proceedings of the International Conference on Dependable Systems and Networks, Los Alamitos 2005, S. 560 ff.

[23] Grottke, M. und Trivedi, K.S., Truncated non-homogeneous Poisson process models – Properties and performance, OPSEARCH 42 (2005), S. 310 ff.

[24] Iannino, A., Software reliability theory, in: Marciniak, J. (Hrsg.), Encyclopedia of software engineering, New York 1994, S. 1223 ff.

[25] International Standard ISO/IEC 15504, Information technology – Process assessment, Parts 1–6, Genf 2003–2006.

[26] Isoda, S., A criticism on the capture-and-recapture method for software reliability assurance, The Journal of Systems and Software 43 (1998), S. 3 ff.

[27] Jelinski, Z. und Moranda, P., Software reliability research, in: Freiberger, W. (Hrsg.), Statistical computer performance evaluation, New York 1972, S. 465 ff.

[28] Johnson, N.L. und Kotz, S., Urn models and their application, New York 1977.

[29] Kuo, L. und Yang, T.Y., Bayesian computation for nonhomogeneous Poisson processes in software reliability, Journal of the American Statistical Association 91 (1996), S. 763 ff.

[30] Lakey, P.B. und Neufelder, A.M., System and software reliability assurance guidebook, Rome Laboratory, Rome 1997.
Verfügbar unter http://www.softrel.com/notebook.zip (Abruf am 13.01.2011).

[31] Langberg, N. und Singpurwalla, N.D., A unification of some software reliability models, SIAM Journal of Scientific and Statistical Computing 6 (1985), S. 781 ff.

[32] Ledoux, J., Software reliability modeling, in: Pham, H. (Hrsg.), Handbook of reliability engineering, London 2003, S. 213 ff.

[33] Littlewood, B., Stochastic reliability growth: A model for fault-removal in computer-programs and hardware-design, IEEE Transactions on Reliability 30 (1981), S. 313 ff.

[34] Littlewood, B. und Verrall, J.L., A Bayesian reliability growth model for computer software, Journal of the Royal Statistical Society, series C 22 (1973), S. 332 ff.

[35] Littlewood, B. und Verrall, J.L., Likelihood function of a debugging model for computer software reliability, IEEE Transactions on Reliability 30 (1981), S. 145 ff.

[36] Lyu, M.R. (Hrsg.), Handbook of software reliability engineering, New York 1996.

[37] Lyu, M.R. und Nikora, A., A heuristic approach for software reliability prediction: The equally-weighted linear combination model, Proceedings of the 1991 IEEE International Symposium on Software Reliability Engineering, Los Alamitos 1991, S. 172 ff.

[38] Lyu, M.R. und Nikora, A., CASRE – A computer-aided software reliability estimation tool, Proceedings of the 1992 IEEE Computer-Aided Software Engineering Workshop, Los Alamitos 1992, S. 264 ff.

[39] Malaiya, Y.K. und Denton, J.A., What do the software reliability growth model parameters represent?, Technical report CS-97-115, Computer Science Department, Colorado State University, Fort Collins 1997.

[40] Mazzuchi, T.A. und Singpurwalla, N.D., Software reliability models, in: Krishnaiah, P.R. und Rao, C.R. (Hrsg.), Handbook of statistics, Vol. 7, Amsterdam 1988, S. 73 ff.

[41] Mazzuchi, T.A. und Soyer, R., A Bayes empirical-Bayes model for software reliability, IEEE Transactions on Reliability 37 (1988), S. 248 ff.

[42] Misra, P.N., Software reliability analysis, IBM Systems Journal 22 (1983), S. 262 ff.

[43] Moranda, P.B., Event-altered rate models for general reliability analysis, IEEE Transactions on Reliability 28 (1979), S. 376 ff.

[44] Munson, J.C. und Khoshgoftaar, T.M., Software metrics for reliability assessment, in: Lyu, M.R. (Hrsg.), Handbook of software reliability engineering, New York 1996, S. 493 ff.

[45] Musa, J.D., Software reliability engineering, New York 1999.

[46] Musa, J.D. und Okumoto, K., A logarithmic Poisson execution time model for software reliability measurement, Proceedings of the Seventh International Conference on Software Engineering, Piscataway 1984, S. 230 ff.

[47] Musa, J.D., Iannino, A. und Okumoto, K., Software reliability: Measurement, prediction, application, New York 1987.

[48] Nelson, E., Estimating software reliability from test data, Microelectronics and Reliability 17 (1978), S. 67 ff.

[49] Ohba, M., Software reliability analysis models, IBM Journal of Research and Development 28 (1984), S. 428 ff.

[50] Okamura, H., Sakoh, T. und Dohi, T., Variational Bayesian approach for exponential software reliability model, Proceedings of the Tenth IASTED International Conference on Software Engineering and Applications, Anaheim 2006, S. 82 ff.

[51] Okamura, H., Grottke, M., Dohi, T. und Trivedi, K.S., Variational Bayesian approach for interval estimation of NHPP-based software reliability models, Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Los Alamitos 2007, S. 698 ff.

[52] Pham, H., System software reliability, London 2010.

[53] Piwowarski, P., Ohba, M. und Caruso, J., Coverage measurement experience during function test, Proceedings of the 15th International Conference on Software Engineering, Los Alamitos 1993, S. 287 ff.

[54] Schick, G.J. und Wolverton, R.W., An analysis of competing software reliability models, IEEE Transactions on Software Engineering 4 (1978), S. 104 ff.

[55] Schneidewind, N.F. und Keller, T.W., Applying reliability models to the space shuttle, IEEE Software 9 (Juli 1992), S. 28 ff.

[56] Shantikumar, J.G., A general software reliability model for performance prediction, Microelectronics and Reliability 21 (1981), S. 671 ff.

[57] Singpurwalla, N.D. und Wilson, S.P., Software reliability modeling, International Statistical Review 62 (1994), S. 289 ff.

[58] Singpurwalla, N.D. und Wilson, S.P., Statistical methods in software engineering: Reliability and risk, New York 1999.

[59] Snyder, D.L. und Miller, M.I., Random point processes in time and space, New York 1991.

[60] Takahashi, M. und Kamayachi, Y., An empirical study of a model for program error prediction, Proceedings of the Eighth International Conference on Software Engineering, Los Alamitos 1985, S. 330 ff.

[61] Thayer, T.A., Lipow, M. und Nelson, E.C., Software reliability, Amsterdam 1978.

[62] Wald, A., Sequential analysis, New York 1947.

[63] Xie, M., Software reliability modelling, Singapore 1991.

[64] Xie, M. und Hong, G.Y., Software reliability modeling, estimation and analysis, in: Balakrishnan, N. und Rao, C.R. (Hrsg.), Handbook of statistics, Vol. 20, Amsterdam 2001, S. 707 ff.

[65] Yamada, S., Software reliability models, in: Osaki, S. (Hrsg.), Stochastic models in reliability and maintenance, Berlin 2002, S. 253 ff.

[66] Yamada, S., Ohba, M. und Osaki, S., S-shaped reliability growth modeling for software error detection, IEEE Transactions on Reliability 32 (1983), S. 475 ff.

[67] Yamada, S., Hishitani, J. und Osaki, S., Software-reliability growth with a Weibull test-effort: A model & application, IEEE Transactions on Reliability 42 (1993), S. 100 ff.

[68] Yang, B. und Xie, M., A study of operational and testing reliability in software reliability analysis, Reliability Engineering and System Safety 70 (2000), S. 323 ff.

[69] Zhang, X. und Pham, H., An analysis of factors affecting software reliability, The Journal of Systems and Software 50 (2000), S. 43 ff.