Modeling and Predicting Software Failure Costs

Michael Grottke*‡ and Christian Graf‡
*Department of Statistics and Econometrics, University of Erlangen-Nürnberg
Lange Gasse 20, D-90403 Nürnberg, Germany
Michael.Grottke@wiso.uni-erlangen.de
‡imbus AG
Kleinseebacher Straße 9, D-91096 Möhrendorf, Germany
{Michael.Grottke, Christian.Graf}@imbus.de

***Abstract*—For software, the costs of failures are not clearly understood. Often, these costs disappear in the costs of testing, the general developments costs, or the operating expenses.**

In a general manufacturing context, the British Standard BS-6143-2:1990 classifies quality-related costs into prevention costs, appraisal costs, and failure costs. It furthermore recommends to identify the activities carried out within each of these categories, and to measure the costs connected with the activities. The standard thus presents a framework for recording and structuring costs once they have occurred.

In this paper, we propose an approach for structuring the information on internal and external software failure costs such that their development over time can be represented by stochastic models. Based on these models, future failure costs can be predicted. In two case studies we show how the approach was applied in an industrial software development project.

***Keywords*-activity-based costing; continuous-time Markov chain; cost prediction; Markov reward model; software failure costs;**

I. Introduction

Crucial questions in software testing include the following: How can testing with a limited budget ensure that a computing system will reliably provide the required services? What is the impact of a failure incident on the supplier and the user(s) of the software system? These questions highlight the need for extending the understanding of software dependability and software quality from a mere technical viewpoint to its economic dimension.

In 2002, a large-scale study prepared for the National Institute of Standards and Technology [1] showed that an economically meaningful assessment of software quality issues can be provided if the costs of software testing are compared with the financial consequences of failures. The report demonstrated that improvements in the infrastructure for testing dependable systems require a clear understanding of the costs incurred by failure occurrences, as well as by the detection and the removal of faults.

However, the identification of failure costs is usually limited to the separate recording of the costs of fault removal [2]. Moreover, direct expenses of a fault (like the costs incurred for detecting and correcting the fault in the software code) and general expenses (like the costs of testing, or the costs of maintenance) are often not distinguished. When calculating the "cost per fault" metric (a.k.a. "cost per defect"), the accumulated costs are thus divided by the total number of faults removed. Using this approach, high quality results in high costs per fault, because the fixed costs are allocated to a small number of faults. Jones [3, p. 483] therefore harshly criticizes cost per defect as "one of the worst and most foolish metrics ever devised." Obviously, direct and general costs need to be separated.

For this reason, the British Standard BS-6143-2:1990 [4] stresses the importance of allocating quality costs to activities. Following concepts of Total Quality Control [5], the standard classifies quality-related costs into costs for prevention, appraisal, and failure; the approach is thus termed the prevention, appraisal and failure (PAF) model.

Prevention costs can be considered investments to reduce future appraisal and failure costs. They include costs for the review and verification of design, for quality training, and for quality auditing.

Appraisal costs are incurred for initially checking whether a product meets its quality requirements. Among the examples for this cost type are costs for inspections and reviews.

Failure costs are subdivided into internal and external failure costs. Internal failure costs arise when inadequate quality of the product is discovered before its ownership has been transferred from the supplier to the purchaser. They include the costs for failure analysis, rework and repair, as well as for reinspection and retesting. External failure costs are caused by inadequate quality discovered after transfer of ownership. Prominent examples are costs incurred for the investigation of complaints (i.e., support), repairs, concessions, and product liability, as well as costs due to the loss of sales. Translated into the language of software engineering, internal failure costs arise when a fault is detected in the software or a related work product during verification and validation activities, especially in the software testing phase. The financial consequences of failure occurrences during operational software usage are external failure costs. An assessment of both types of failure costs can provide valuable information on how to improve processes related to prevention and appraisal in an organization.

Approaches combining the PAF model with activity-based costing are well-known in financial controlling of manufacturing industries [6]. Following Ittner [7], Karg and Beckhaus [8] suggest an activity-based approach to measuring the costs in the context of software quality assurance.

The vast majority of models allowing the estimation of software costs are regression models [9]. Based on an assessment of certain key influence factors, these models predict either costs or the number of faults; see e.g. [2] and [10]. Moreover, there is also a variety of approaches using software reliability growth models for modeling quality costs [11], [12]. Such approaches offer for example the possibility to estimate the optimal release time, or to carry out a risk assessment of external failure costs. Their drawback is that the preconditions for the application of software reliability growth models must be met. This may prohibit their adoption in contexts where systematic testing is employed [13].

The cost optimization model by Wagner [14] is based on the PAF approach, although it neglects prevention costs. In this model, internal failure costs (as the costs related to the removal of faults) and external failure costs (as costs of fault removal on the one hand, and costs incurred by failure occurrences on the other hand) are considered explicitly. Although the costs of each fault may depend on the order in which the faults are detected, a method for determining these varying costs is not in the scope of [14].

With respect to the determination of the activities caused by failures, as well as their costs, the approach proposed in this paper is similar to the ones suggested in [7] and [8]. However, we allow the relationship between failure occurrences (as well as other events) and activities to be more complex than in [8] and in other models. For example, activities like the release of a service pack may be executed only after a number of failures have been experienced. Moreover, activities may not be triggered for sure, but only with a certain probability.

In our approach, information is structured in a way allowing one or more continuous-time Markov chain (CTMC) models to be built. Such a model captures the dynamics of failure occurrences and other cost-relevant events. Among the outputs that can be generated is the predictive probability mass function of costs incurred until some future point in time.

The paper is structured as follows: In Section II, we describe the general approach of assessing and structuring the information in a way suitable for model building. Section III illustrates the application of the method by means of two case studies. While the first case study is a rather simple example of analyzing internal failure costs (see Section III-A), the second one, dealing with external failure costs shows how even complex dependencies between failure occurrences and activities can be assessed and modeled (see Section III-B). Section IV concludes the paper and gives a brief outlook on our future research.

II. The general approach

We propose the following six-step procedure for assessing, structuring, and modeling software failure costs:

1. Define activities and assign costs.

In a first step, the activities that may be required in response to a failure occurrence need to be listed, together with the costs of carrying out each of them once. Techniques that may help in the identification of activities include brainstorming, flowcharting, and interviewing [7]. While the costs incurred for performing an activity include material, in the software development context labor costs usually represent the largest part. If employees record the time spent on the various activities, then these accounting records can be combined with information on the frequency with which each activity was carried out, to derive the efforts in man-days or man-hours for performing each activity once. As an alternative, experts (like test managers) can come up with sustainable estimates. Experts can also help to validate information extracted from the accounting system.

The costs incurred by an activity may depend on the process history, like the number of times the activity has been performed before. For instance, the costs caused by a failure may be a function of the number of faults detected previously, similar to the approach by Wagner [14].

Given the process history, the costs assigned to an activity are assumed to be deterministic. The activities should therefore be defined in a way lending itself to this assumption. For example, the costs of the activity “fix fault” may show a lot of random fluctuation because they depend on specifics of the respective fault; a mere coding fault is generally easier to fix than a fault that was introduced in the requirements phase of software development. One possible solution is to define one activity for each fault type (e.g., “fix requirements fault”, “fix coding fault”). Another feasible approach, which will be illustrated in the first case study, is to identify sub-activities of the activity “fix fault”.

2. Define activity groups.

Certain circumstances may require not one, but multiple activities to be carried out. For example, if a fault in the documentation is detected, this fault has to be reported and fixed, and the fix then needs to be verified. Activities caused by the same event (whether or not they are performed simultaneously) are combined in activity groups. Since some activities may have to be carried out more than once, the number of “repeated” executions of an activity in an activity group has to be specified; this number may depend on the process history. If, for example, a software developer fixes known faults in a software product only when preparing a service pack, then the number of times that the activity “fix fault” is performed is equal to the number of faults detected since the last service pack. To simplify the overall structure, we assume that even activities that are carried out alone upon

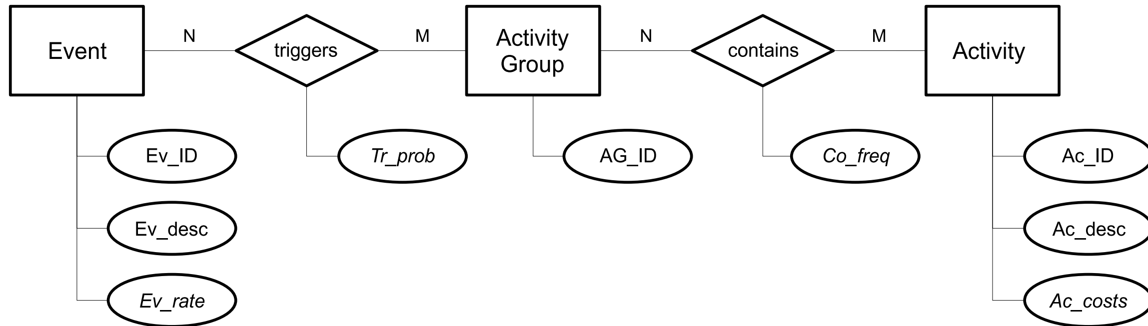

Figure 1. Entity relationship model of the elements used in the general approach

an event are assigned to an activity group consisting of this one activity.

3. Define events.

As mentioned before, activities are performed in response to events, like the detection of a fault. For each of the activity groups defined in the last step, we now need to identify one or more events triggering this activity group. Furthermore, the rate with which each event occurs (i.e, the expected number of occurrences per unit time) has to be specified. This information can be gained from fault databases, customer support records, etc. Again, the occurrence rates may depend on the process history. This allows us, for example, to account for an increasing reliability of the software product by making the fault detection rate a decreasing function of the number of faults detected (and fixed) before. In the following, we will implicitly assume that given the previous process history, different events occur independently of each other. Suitable definition of activities, activity groups, and events can help to ensure that this assumption is a good approximation to reality. In general, the identification of events requiring adequate response may guide the refinement of the activities and activity groups defined in the previous steps.

4. Define trigger probabilities.

The occurrence of an event may sometimes, but not always, cause the activities in one activity group to be carried out. For each combination of an event and an activity group, we therefore allow the specification of a trigger probability between zero and one; this probability may depend on the previous history of the process. Such probabilities can be based on recorded data (e.g., by using the relative frequency with which an activity group was performed upon the occurrence of an event). Since the trigger probabilities are related to the policies of the company (e.g., "how likely is it that we will distribute a service pack after a fault of a certain type has been reported?"), experts like the support personnel should be able to provide realistic values even if data are not available.

A convenient way of representing the elements used in our approach is the entity relationship model shown in Fig. 1. We employ the classic notation by Chen [15], including attributes for both entity sets and relationship sets. Besides a unique identifier (Ev_ID), events have a description (Ev_desc) and an occurrence rate (Ev_rate). Each event can trigger one or multiple activity groups; likewise, an activity group may be triggered by several events. A triggering probability (Tr_prob) is attributed to each "triggers" relationship between one event and one activity group. For our purposes, we only require an activity group to have a unique key (AG_ID). It could be further characterized by a description, etc. Each activity group needs to contain at least one activity. The attribute Co_freq gives the number of times that a certain activity contained in a specific activity group is carried out after this activity group has been triggered. Like events, each activity has an identifier (Ac_ID) and a description (Ac_desc); moreover, it is assigned the costs for carrying out the activity once (Ac_costs). Each activity may be contained in more than one activity group.

In Figure 1, Ev_rate, Tr_prob, Co_freq, and Ac_costs are in italics, indicating that these attributes may depend on the process history. More specifically, they are allowed to depend on the following pieces of information:

- "Global" counters of the occurrences of events since the beginning of the prediction period (e.g., the total number of failure occurrences).
- "Local" counters of the occurrences of events (e.g., the number of failure occurrences since the last service pack was shipped).
- "Global" counters of the triggering of activity groups since the beginning of the prediction period (e.g., the total number of service packs released).
- "Local" counters of the triggering of activity groups (e.g., assuming that patches are distributed only after a number of faults of a certain severity have been found, the number of patches distributed since the last service pack was shipped).

In a first step, the dependence of an attribute on the respective history information can be formulated in words; however, it needs to be possible to translate this formulation into a mathematical equation. We will see examples for that in the second case study.

The entity relationship model suggests that the elements in our approach as well as their relationships can conveniently be represented in tables. In fact, this is how we will present the information in the case studies.

5. Partition set of events for independent modeling.

The information gathered and structured in the four preceding steps suffices for building a continuous-time Markov chain (CTMC) model [16] that captures the occurrence of events as well as the triggering of activity groups over time. Basically, it is possible to consider all events and activity groups in one comprehensive model. However, the model size (i.e., the number of states in the CTMC) increases dramatically with each additional activity group and event included, slowing down or even precluding calculations. It is therefore advisable to build not one comprehensive model, but as many separate submodels as possible. The information collected during the preceding steps helps in deciding whether or not a set of events can be separated into an independent CTMC model.

Consider a subset of the events defined before, for which this question is to be answered. We use the term "set of elements induced" by this subset to refer to the set of elements consisting of

1) the events under consideration;
2) all "triggers" relationships that these events are involved in, and the related activity groups;
3) all "contains" relationships that these activity groups are involved in, as well as the related activities.

The subset of events can be modeled separately if the following two conditions are met:

a) The rates Ev_rate, the probabilities Tr_prob, the frequencies Co_freq, and the costs Ac_costs of all elements in the induced set of elements do not depend on the history of any element not included in the induced set of elements; and
b) The histories of the events and triggers in the induced set of elements do not influence any rates, probabilities, frequencies, or costs of elements not included in the induced set of elements.

Based on this criterion, the set of all events defined should be partitioned into as many subsets as possible.

6. Build and use CTMC models.

For each of the subsets into which the set of all events was partitioned in step 5, an individual CTMC model is now constructed. The states in such a CTMC count the number of times that activity groups have been triggered, as well as the occurrences of relevant events. The model complexity thus depends on the number of events and "triggers" relationships, as well as on the pieces of information influencing the rates, probabilities, frequencies, and costs in the set of elements induced by the respective subset of events.

Each CTMC model can for example be used to derive the probability mass function of the number of times a certain activity group will be triggered within a given prediction period. Moreover, by assigning cost values (directly following from the information assessed during steps 1 to 4) to the states, the models are extended into Markov reward models [16]. Based on these models, the probability mass function of costs incurred in the prediction period can be calculated.

III. Case studies

Both case studies described in this section address the failure costs of the same platform-independent commercial off-the-shelf software product developed by an IT company with approximately 200 employees. The software features a client-server architecture and interfaces to several external applications, especially databases. Nine developers and five employees in the test and support team are in charge of this software product. In order to protect confidential information all cost values and event rates have been normalized.

A. Modeling internal failure costs

The goal of the first case study was to model and predict the internal failure costs incurred during the development of a standard release of the software product. Fault data collected during the first two months of the observation period served as a basis for the prediction of the internal failure costs for the remaining three months before release. For implementing *step 1* of the general approach described in Section II, a detailed list of (sub-)activities potentially triggered upon the detection of a fault was compiled based on the current knowledge of the relevant processes. The activities identified are shown in Table I. The related costs were estimated by the project manager and the team leads of the test and development teams. Rejected fault reports (false positives) were considered appraisal costs, not failure costs; we therefore excluded them from our analysis.

In *step 2*, the definition of activity groups, we first clarified under which circumstances which activities are

Table I
Activities

Ac_ID	Ac_desc	Ac_costs
Ac1	Report failure	0.083
Ac2	Analyze report	0.500
Ac3	Fix requirements	0.167
Ac4	Fix design	0.750
Ac5	Fix implementation	3.500
Ac6	Fix documentation	0.125
Ac7	Verify fix	0.167

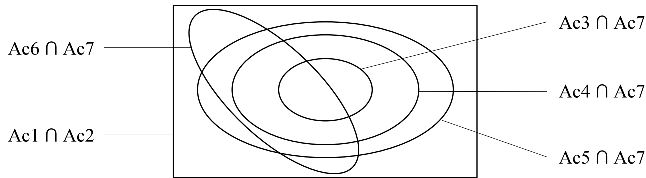

Figure 2. Venn diagram of activities

jointly carried out. For any observed failure a report has to be written by the tester (Ac1) and analyzed by a developer (Ac2). The result of the analysis either pinpoints the fault that caused the failure, or it results in the decision to treat the report as a suggestion, which may be dealt with in a future release. In the latter case, no further activities are necessary. In the former case, however, some kind of fix and thus its subsequent verification (Ac7) is required. On the one hand, either Ac5, or Ac5 and Ac4, or Ac5, Ac4 and Ac3 may be carried out: A fault in a requirement always involves fixing the requirement and the related design; a fault in the design always involves a fix of the implementation. On the other hand, and independently of the other fixes necessary, a fix of the documentation (Ac6) might be required. Based on the set of all faults the Venn diagram in Fig. 2 is a graphical representation of these relationships between the necessary activities. Each intersection between the sets in this Venn diagram was identified as an activity group. The set of all activity groups thus forms a partition of the set of all faults.

Table II displays all "contains" relationships between activity groups and activities. Since the activities in each group are carried out exactly once when the group is triggered all entries in the right column are equal to one. The second case study in Section III-B will demonstrate that these entries may differ from this value.

Table II
"CONTAINS" RELATIONSHIPS

AG_ID	Ac_ID	Co_freq
AG1	Ac1	1
	Ac2	1
AG2	Ac1	1
	Ac2	1
	Ac5	1
	Ac7	1
AG3	Ac1	1
	Ac2	1
	Ac4	1
	Ac5	1
	Ac7	1
AG4	Ac1	1
	Ac2	1
	Ac3	1
	Ac4	1
	Ac5	1
	Ac7	1
AG5	Ac1	1
	Ac2	1
	Ac6	1
	Ac7	1

AG_ID	Ac_ID	Co_freq
AG6	Ac1	1
	Ac2	1
	Ac5	1
	Ac6	1
	Ac7	1
AG7	Ac1	1
	Ac2	1
	Ac4	1
	Ac5	1
	Ac6	1
	Ac7	1
AG8	Ac1	1
	Ac2	1
	Ac3	1
	Ac4	1
	Ac5	1
	Ac6	1
	Ac7	1

Table III
EVENTS

Ev_ID	Ev_desc	Ev_rate
Ev1	Suggestion by tester	$\lambda_1 = 7.33$
Ev2	Detection of implementation fault not requiring change in documentation	$\lambda_2 = 11.47$
Ev3	Detection of design fault not requiring change in documentation	$\lambda_3 = 1.18$
Ev4	Detection of requirements fault not requiring change in documentation	$\lambda_4 = 1.69$
Ev5	Detection of mere documentation fault	$\lambda_5 = 2.54$
Ev6	Detection of implementation fault requiring change in documentation	$\lambda_6 = 5.07$
Ev7	Detection of design fault requiring change in documentation	$\lambda_7 = 0.52$
Ev8	Detection of requirements fault requiring change in documentation	$\lambda_8 = 0.75$

Each of the activity groups obviously represents the activities required upon the detection of a certain type of fault. Assuming that different fault types are detected independently of each other, we therefore identified one triggering event for each of the activity groups AG1 to AG8, namely the detection of a fault of the respective type. These events defined following *step 3* of the general approach are listed in Table III. The fault detection rates for each trigger-event were obtained from a detailed analysis of the fault data collected during the first two months of tests.

The structure of the "triggers" relationships as well as the trigger probabilities identified following *step 4* of the general approach are very simple: Event Evi triggers activity group AGi with probability one ($i = 1, 2, \dots, 8$); see Table IV.

None of the rates, probabilities, frequencies and costs defined in this case study depend on the process history. Therefore, the set of elements induced by each individual event meets the conditions a) and b) formulated in *step 5*. As

Table IV
"TRIGGERS" RELATIONSHIPS

Ev_ID	AG_ID	Tr_prob
Ev1	AG1	1
Ev2	AG2	1
Ev3	AG3	1
Ev4	AG4	1
Ev5	AG5	1
Ev6	AG6	1
Ev7	AG7	1
Ev8	AG8	1

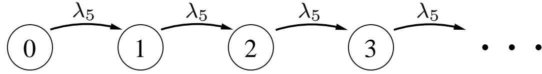

Figure 3. Continuous-time Markov chain for event Ev5

a consequence, the occurrence of each of these events can be modeled separately. Since the occurrence rates are constant and the trigger probabilities are one, the CTMC model for Evi counts how often the related activity group AGi has been triggered; it features the structure shown in Fig. 3 for $i = 5$. The number of times an activity group will be triggered in a given prediction period thus follows a Poisson distribution; from this distribution, the distribution of the costs incurred by this activity group is easily calculated. The distribution of the total internal failure costs is derived as the convolution of the cost distributions for all activity groups. From this predictive distribution measures like the mean, the variance, the median, and further quantiles can be computed.

Based on the data collected during the first two months of testing, we calculated the predictive distribution of internal failure costs for several future points in time. Fig. 4 compares the development of the 95% prediction interval and the median of the predictive distribution with the actual cumulative costs incurred in the subsequent three months of testing before release. While up to about eight units of additional test effort the actual costs were within the limits of the confidence interval, the last two observations were smaller than its lower bound. Further analysis of the fault data revealed a shift in the fault type proportions: in the late testing phase, suggestions exceeded any other fault type. A potential reason for this phenomenon might be the tendency to under-evaluate the importance of faults shortly before a release date. However, our investigation has shown that a mere one percent of these unfixed faults were later reported by customers. The increasing proportion of suggestions therefore does indeed seem to indicate a growth in the reliability of the system during test.

This discussion shows that our specification of constant event rates was overly simplistic: While the occurrence rate of suggestions increased over time, the rates with which other fault types were detected became smaller. However, as pointed out before, the general approach proposed in Section II does allow the formulation of event rates that depend on the process history. In the second case study we will make use of this property.

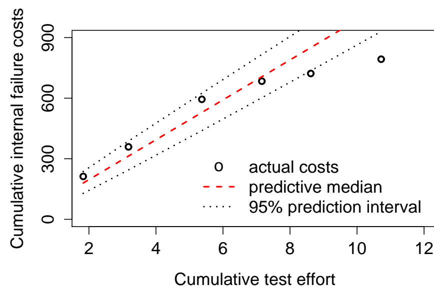

Figure 4. Predicted and actual internal failure costs

B. Modeling external failure costs

The second case study was aimed at identifying cost drivers of external failure costs incurred for the software product, at determining their magnitude, as well as at making predictions for the current release. To this end we used information collected for the previous release both during testing and in the field, as well as in the testing phase of the current version.

Interviews with experts revealed three different kinds of reactions to faults reported by customers (thus classified as faults of type 1, 2, and 3): Type 1 faults have severe consequences potentially affecting many customers; they require the immediate distribution of a service pack. Due to the high costs incurred by such a service pack, costs of further activities (like customer support for a type 1 fault) can be neglected. Type 2 faults do not necessitate the immediately distribution of fix; when a type 2 fault is reported, the development team fixes it in the current main branch of the software. However, when a service pack needs to be built, these fixes are ported from the main branch. While customers tolerate a limited number of known type 2 faults in the software, the larger the number of open issues, the more vigorously customers will demand a service pack. Type 3 faults are critical, but they only affect one customer. If such a fault is reported, the customer is provided with a patch. In addition to the detection of these three types of faults, customers address the support team when further issues arise, e.g., misunderstandings of the documentation, or problems with respect to the usability of the software product. Beyond customer support these issues do not require any immediate action.

Table V lists the identified activities together with cost estimates made by the experts, thus completing *step 1* of the general approach.

Based on the above description, we also carried out *step 2* and defined the activity groups shown in Table VI. Note that

Table V
ACTIVITIES

Ac_ID	Ac_desc	Ac_costs
Ac1	Build and distribute service pack	200
Ac2	Customer support for type 2 fault	1
Ac3	Analyze and fix type 2 fault in main branch	4
Ac4	Adapt and test fix for type 2 fault in service pack	4
Ac5	Customer support for type 3 fault	1
Ac6	Build and distribute patch	12
Ac7	Customer support for further issue	1

Table VI
"CONTAINS" RELATIONSHIPS

AG_ID	Ac_ID	Co_freq
AG1	Ac1	1
	Ac4	j
AG2	Ac2	1
	Ac3	1
AG3	Ac5	1
	Ac6	1
AG4	Ac7	1

each service pack includes ported fixes for all j type 2 faults detected since the last release. j is a local event counter, a part of the process history.

An obvious choice for *step 3* is to identify the detection of faults of the various types, as well as the emergence of further issues as events; see Table VII. The event rates were estimated based on the available data of the previous release and the current test phase. All rates are expressed in terms of the number of occurrences per unit of software execution time. Since occurrences of Ev1 are rare, data on this event were scarce. λ_1 was therefore estimated by the field occurrence rate of failures caused by type 1 faults for the previous release.

Unlike type 3 faults, type 2 faults had shown a decreasing detection rate during the operational phase of the previous release, which could be modeled well with Moranda's geometric de-eutrophication software reliability model [17]. We assumed that for the current release the occurrence rate of Ev2 had the functional form related to this model, shown in Table VII. In this function, the global event counter l represents the total number of type 2 faults detected since the prediction origin. As geometric rate r we chose 0.9798, the estimate obtained based on the field data of the previous release. The initial failure rate ϕ was estimated as the residual detection rate of type 2 faults during the current test phase, multiplied by the rate adaptation factor (i.e., the detection rate during the field divided by the detection rate during testing) established for this fault type based on the previous release; the value obtained was $\phi = 4.3991$. Similarly, λ_3 was estimated by multiplying the (average) rate with which type 3 faults were detected during the current test phase with the rate adaptation factor for type 3 faults in the previous release.

Table VII
EVENTS

Ev_ID	Ev_desc	Ev_rate
Ev1	Detection of a type 1 fault	$\lambda_1 = 0.0115$
Ev2	Detection of a type 2 fault	ϕr^l
Ev3	Detection of a type 3 fault	$\lambda_3 = 0.0817$
Ev4	Emergence of a further issue	$\lambda_4 = 3.4511$

Table VIII
"TRIGGERS" RELATIONSHIPS

Ev_ID	AG_ID	Tr_prob
Ev1	AG1	1
Ev2	AG2	1
	AG1	$p_j = \max\left(0, \frac{j-z}{k-z}\right)$
Ev3	AG3	1
Ev4	AG4	1

The trigger probabilities defined in *step 4* are displayed in Table VIII. The probability p_j that the detection of a type 2 fault triggers a service pack accounts for the aforementioned customer behavior: For up to z type 2 faults since the last service pack, the probability that a service pack is demanded is zero. The probability then increases linearly with each detected type 2 fault and reaches one when $k > z$ type 2 faults have been detected. It depends on the same local event counter j as the "contains" relationship between AG1 and Ac4. For our specific case study, experts determined $z = 45$ and $k = 60$ to be reasonable values.

Identifying the "sets of elements induced" for Ev1, Ev2, Ev3, and Ev4 in *step 5* shows that the sets $\{$Ev1, Ev2$\}$, $\{$Ev3$\}$ and $\{$Ev4$\}$ fulfill the conditions a) and b). The events Ev3 and Ev4 can thus be modeled separately in the same manner as the events in the first case study. However, the "contains" relationship between AG1 and Ac4, contained in the "set of elements induced" by Ev1, depends on j, the number of Ev2 occurrences since the last service pack. Thus, Ev1 and Ev2 must be modeled together.

Fig. 5 gives an idea of the CTMC for events Ev1 and Ev2, built in *step 6*; note that due to the three-dimensional structure of the CTMC only some of the transitions to higher levels can be shown in the two-dimensional diagram. Each state (i, j, l) is characterized by the total number of service packs i as well as the number of type 2 faults detected since the last service pack j and in total l. Based on this information, the corresponding number of times that the activities Ac1 to Ac4 were executed and thus the external failure costs related to each state are easily calculated; these costs can be attached to a state as its reward rate. In Fig. 5 the states are arranged in levels. Level v contains all states for which $l - j$, the number of type 2 faults detected previous to the last service pack, equals v.

At time 0, the prediction origin, the process starts in state $(0, 0, 0)$. The transient state probabilities (and based on them the probability mass function of the external failure costs due to events Ev1 and Ev2) after a given prediction period are obtained numerically via the uniformization (a.k.a. randomization) algorithm [18], [19]. We implemented this algorithm in R [20]. Employing the Matrix package [21] allowed us to make use of the fact that the transition probability matrix of the discrete-time Markov chain subordinated to the CTMC

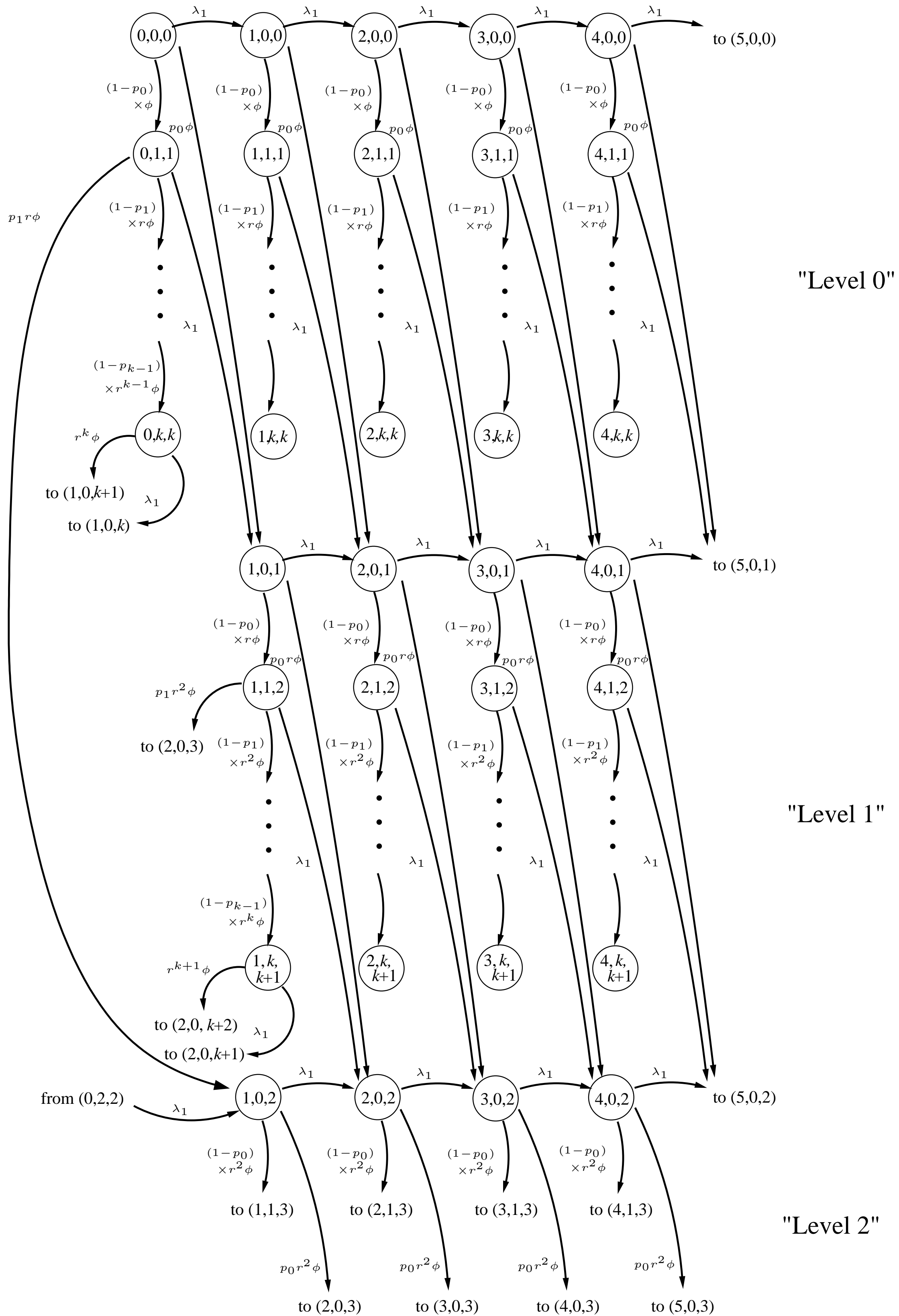

Figure 5. Continuous-time Markov chain for events Ev1 and Ev2

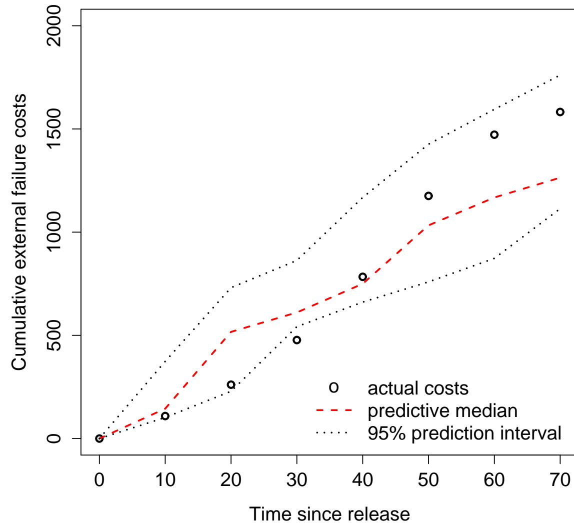

Figure 6. Predicted and actual external failure costs

is sparse. We were thus able to obtain results even for a huge number of states, truncating the CTMC in Fig. 5 in regions that are highly unlikely to be reached. For example, the calculations shown below are based on a CTMC for events Ev1 and Ev2 with 208986 states; only 832520 entries in the related 208986×208986 transition probability matrix are larger than zero.

The predictive distribution of the total external failure costs is again obtained as the convolution of the cost distributions following from the three CTMC submodels for the sets {Ev1, Ev2}, {Ev3}, and {Ev4}.

After the release of the current software version, the actual total external failure costs were determined at seven evenly-spaced moments in calendar time. Fig. 6 shows their development together with the related predictive medians and 95% prediction intervals calculated based on the information available before release. All but one observation lie within the interval bounds.

From Fig. 6, the shape of the predictive probability mass functions does not become clear. We therefore display the predictive probability mass function for the total external failure costs incurred after 50 calendar time units in Fig. 7. The five peaks discernible in this multimodal distribution correspond to the creation of 0, 1, 2, 3, and 4 service packs, respectively.

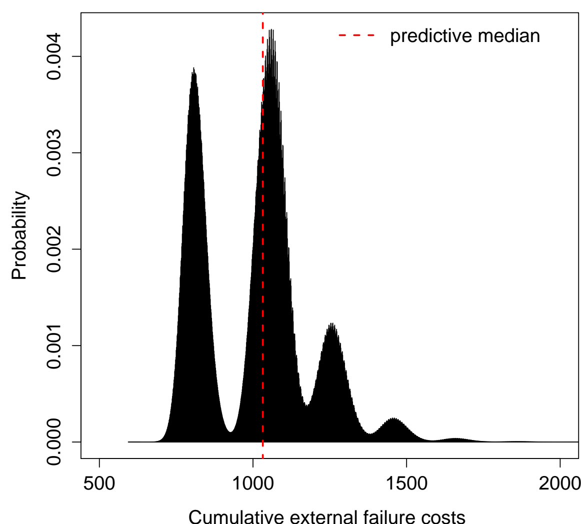

Figure 7. Predictive probability mass function of cumulative external failure costs after 50 calendar time units

IV. Conclusions

The possibility to model failure costs is an asset for any test or release manager. In this paper, we presented a structured approach to assessing and modeling costs connected with software failures. By relating events to activity groups, it helps identify cost drivers and understand the dynamics of cost-relevant events. Its prediction capabilities allow an improved cost-driven planning and controlling of software projects. The approach can help in adequately allocating resources.

In our future research, we will investigate further ways to specifying adequate model parameters in the absence of data from previous releases of the software product. For example, classifying the faults detected in the current test phase may help draw conclusions about the fault detection rates during the remainder of testing, or in the field.

Moreover, we are planning to combine our approach with models for appraisal costs, to support justified release decisions based on the minimization of expected overall costs.

Acknowledgments

Parts of the work presented in this paper have been funded by the BMBF SE2006 project TestBalance (grant 01ISF08A).

References

[1] RTI, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Gaithersburg, Planning Report 02-3, 2002.

[2] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*. Upper Saddle River: Prentice Hall, 2000.

[3] C. Jones, *Applied Software Measurement – Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[4] "Guide to the economics of quality – Part 2: Prevention, appraisal and failure model," British Standards Institute, London, BS 6143-2:1990, 1990.

[5] A. V. Feigenbaum, "Total quality control," *Harvard Business Review*, vol. 34, no. 6, pp. 93–101, 1956.

[6] A. Schiffauerova and V. Thomson, "A review of research on cost of quality models and best practices," *International Journal of Quality and Reliability Management*, vol. 23, no. 6, pp. 647–669, 2006.

[7] C. Ittner, "Activity-based costing concepts for quality improvement," *European Management Journal*, vol. 17, no. 5, pp. 492–500, 1999.

[8] L. M. Karg and A. Beckhaus, "Modelling software quality costs by adapting established methodologies of mature industries," in *Proc. 2007 IEEE International Conference on Industrial Engineering and Engineering Management*, 2007, pp. 267–271.

[9] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Trans. Software Engineering*, vol. 33, no. 1, pp. 33–53, 2007.

[10] H. K. N. Leung and L. White, "A cost model to compare regression test strategies," in *Proc. Conference on Software Maintenance*, 1999, pp. 201–208.

[11] S. R. Dalal and C. L. Mallows, "When should one stop testing software?" *Journal of the American Statistical Association*, vol. 83, no. 403, pp. 872–879, 1988.

[12] H. Pham, *Software Reliability*. Singapore: Springer, 2000.

[13] M. Grottke, *Modeling Software Failures during Systematic Testing*. Aachen: Shaker Verlag, 2003.

[14] S. Wagner, *Cost-Optimisation of Analytical Software Quality Assurance*. Saarbrücken: VDM Verlag, 2007.

[15] P. P.-S. Chen, "The entity-relationship model – Toward a unified view of data," *ACM Trans. Database Systems*, vol. 1, no. 1, pp. 9–36, 1976.

[16] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed., New York, 2001.

[17] P. Moranda, "Event-altered reliability rate models for general reliability analysis," *IEEE Trans. Reliability*, vol. 28, pp. 376–381, 1979.

[18] D. Gross and D. R. Miller, "The randomization technique as a modeling tool and solution procedure for transient Markov processes," *Operations Research*, vol. 32, no. 2, pp. 343–361, 1984.

[19] A. Reibman and K. S. Trivedi, "Numerical transient analysis of Markov models," *Computers and Operations Research*, vol. 15, no. 1, pp. 19–36, 1988.

[20] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2009, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[21] D. Bates and M. Maechler, *Package 'Matrix'*, Reference manual, version 0.999375-26, 2009. [Online]. Available: http://cran.r-project.org/web/packages/Matrix/Matrix.pdf