Understanding and Predicting Effort in Software Projects

Audris Mockus and David M. Weiss
Avaya Labs Research
Department of Software Technology Research
233 Mt Airy Rd., Basking Ridge, NJ 07920

Ping Zhang
Avaya Labs Research
Department of Data Analysis Research
233 Mt Airy Rd., Basking Ridge, NJ 07920

Abstract

We set out to answer a question we were asked by software project management: how much effort remains to be spent on a specific software project and how will that effort be distributed over time? To answer this question we propose a model based on the concept that each modification to software may cause repairs at some later time and investigate its theoretical properties and application to several projects in Avaya to predict and plan development resource allocation. Our model presents a novel unified framework to investigate and predict effort, schedule, and defects of a software project. The results of applying the model confirm a fundamental relationship between the new feature and defect repair changes and demonstrate its predictive properties.

Key Words and Phrases: software changes, effort estimation, project schedule, defect prediction

1. Introduction

Despite considerable research and practical experience it is still a formidable challenge to understand and predict what happens in a large software project. Even in cases where some parties have a good understanding of the consequences, the business pressures and lack of quantitative evidence often results in misguided effort and time estimation and faulty plans.

We focus on a problem of planning and managing development resource allocation in large software projects. We set out to answer very specific questions that were urgent in several high-profile projects in Avaya:

1. How much effort remains to be spent on a specific software project based on what we know now?
2. How will that effort be distributed over time?

To answer these questions we built a simple model of a software project that describes the current state and possible outcomes of a project based on various sources of information ubiquitous in software projects, thereby decreasing the need for the subjective judgment. This model proved to be flexible enough to answer a wide range of important project planning questions.

Previous work [1, 11, 13] has identified version control and problem tracking databases as a promising repository of information about a software project. We have created methods and tools to retrieve, process, and model such data at the fine level of Modification Requests (individual changes to software) in order to understand the relationships among process/product factors and key outcomes, such as, quality, effort, and interval.

This work proposes ways to use the change data to understand and predict the state of a software project. We propose a model based on the concept that each modification to software may cause repairs at some later time and investigate the model's theoretical properties and report its application to several projects in Avaya. The model assumes the basic premise of change analysis, i.e., that the people involved in a project leave traces of their work in the form of modifications to the artifacts on which they work.

We start from our motivating questions in Section 2. Section 3 describes ways to obtain data on software changes and describes a method to estimate effort for a software change. The model used to compose a project from software changes is introduced in Section 4; Section 5 describes the result of fitting such models to actual projects; Section 6 considers ways to validate these empirical results, and Section 7 outlines steps needed to model other software projects. We conclude with literature review in Section 8 and discussion.

2. Motivation

We set out with the desire to answer the two basic questions stated in 1. These questions arose in two ongoing projects and in two different contexts. In the first project the key question was whether or not the release date would be met. Would there be sufficient time and effort to complete

the features scheduled for the release, including repairing any problems that arose during the development. In the second project, a new release was about to go into the field. The question was how much effort would be needed to support it once it was released. Since resources devoted to repairing problems, whether found in the field or earlier, are resources that cannot be applied to completing features or to work on other projects, the questions were critical for both projects and for other projects that were awaiting resources.

Surprisingly, the planning in both projects did not take into account that a project of larger magnitude might cause more field problems and drain more resources than a smaller project. Because of space constraints we report results only for the first project.

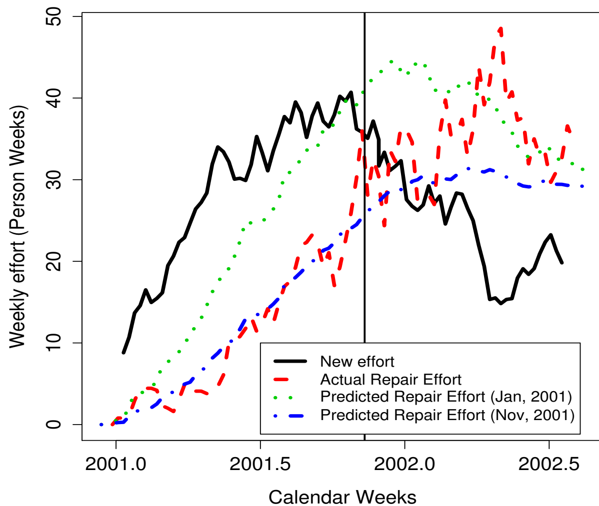

Figure 1. Actual and predicted repair effort.

The first project had an ambitious schedule. However, after a few slips the question of when and what will be complete arose. The project history is illustrated in Figure 1. The figure shows the actual effort for implementing new features (solid line), the effort of repairing defects (dashed line), and two curves of predicted effort with predictions done before the project started (dotted line) and on November 2001 (dash-dot line). The end of November is indicated by a vertical line). The details of the method used to produce the prediction are described below. There was a major release in May and new feature activities in the second quarter of 2002 will be released later via software patches or minor releases.

The predicted repair effort is fairly close to reality. However, the initial prediction at the start of the project, which is based on data from previous projects, is slightly biased because as it turns out, the new project has longer intervals until defect repairs. The revised prediction done in November 2001 uses data from the current project. Our recommendation made to the project in November 2001 was that the May 2002 target for release could not be met unless the development team refocused it's resources to repair of existing defects and away from the development of additional features. The actual effort curves after November 2001 show that the project apparently did exactly that, i.e., reducing the scope of new functionality and focusing on repair.

The next section describes background information on using change management and version control repositories to obtain information on software changes that is used to obtain and predict the the effort distribution in a software project.

3. Background

The basic premise of analyzing software changes is that software is created incrementally through a series of work items, each potentially resulting in a change. Each incremental change is recorded by a version control and, possibly, problem tracking system. The data typically contains a set of attributes such as the following:

- The identity of the person making the change.
- A short comment written by the author of the change.
- The file(s) changed and the lines changed or the file(s) contents before and after the change.

3.1. Work items in software projects

The purpose of the typical work item in a software organization is to make a change to a software entity. Work items range in size from very large work items, such as releases, to very small changes, such as a single delta (modification) to a file. A hierarchy of changes with associated attributes is shown in Figure 2.

The source code of large software products is typically organized into subsystems according to major functionality (e.g., database, user interface, etc.). Each subsystem contains a number of source code files and documentation.

The versions of the source code and documentation are maintained using a version control system (VCS) such as Concurrent Versioning System [4] commonly used for open source software projects, or a popular commercial system, such as ClearCase. We frequently deal with Source Code Control System (SCCS) [15] and its descendants. Version control systems operate over a set of source code files. An *atomic* change, or *delta*, to the program text consists of the lines that were deleted and those that were added in order to make the change. Deltas are usually computed by a file differencing algorithm (such as Unix diff), invoked by the VCS, which compares an older version of a file with the current version. Included with every delta is information

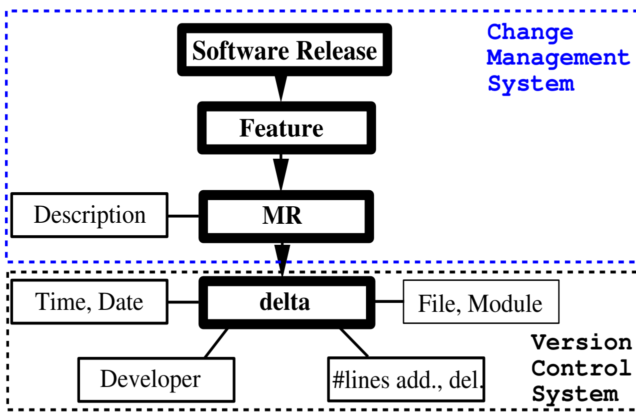

Figure 2. Hierarchy of changes and associated data sources. Boxes with dashed lines define data sources (VCS and CMS), boxes with thick lines define changes, and boxes with thin lines define properties of changes. The arrows define an "is a part of" relationship among changes, e.g., each MR is a part of a feature.

such as the time the change was made, the person making the change, and a short comment describing the change.

In addition to VCS, most projects employ a change request management system (CMS) that keeps track of individual requests for changes, which we call Modification Requests (MRs). Whereas a delta is intended to keep track of lines of code that are changed, an MR is intended to be a change made for a single purpose. Each MR may have many deltas associated with it. Some commonly used problem tracking systems include ClearDDTS from Rational, and the Extended Change Management System (ECMS) [10]. Usually such systems associate a list of deltas with each MR.

Modifications are typically made for one of the following reasons.

- Repairing previous changes that caused a failure during testing or in the field.
- Introducing new features to the existing system.
- Restructuring the code to make it easier to understand and maintain. (An activity more common in heavily modified code, such as in legacy systems.)

To understand the activities occurring in a software development project, it is critical to know which MRs belong to each of these categories. Fortunately, this information is often recorded in CMS systems as a field identifying whether the MR represents a new feature or repairs a problem. Unfortunately, the quality of this classification varies by project. In cases when it is unacceptably low for the purpose of a particular analysis we use an automatic classification technique that uses words in the MR abstract to determine the purpose of the MR [11].

Based on informal interviews in a number of software development organizations within Lucent and Avaya we obtained the following guidelines that are used to divide work into MRs:

1. Work assignments that affect several subsystems (the largest building blocks of functionality) are split into distinct MRs so that each MR affects one subsystem;
2. A work assignment in a subsystem that is too much for one person is further organized into several MRs so that each one could be completed by a single person.

For practical reasons these guidelines are not strictly enforced, so that some MRs cross subsystem boundaries and some have several people working on them.

A group of MRs associated with new software functionality is called a feature. A set of features and repairs constitute a customer delivery, also known as a release. Put another way, each release can be characterized as a base system modified and extended by a set of MRs.

3.2. The value of analyzing changes

The analysis of software changes has a number of distinct benefits that may not be immediately obvious.

- The data collection is nonintrusive, using only existing data and making analysis possible in commercial projects that are usually under intense schedule pressure and do not have time or resources to collect additional data.
- Long history on past projects is available, enabling comparison to what happened in the past and customization and calibration of the methods to the existing environment. Nonetheless, one must be mindful of changes to the environment and application that make comparisons problematic.
- The information is fine grained, at the MR/delta level. Such fine level data collection on a large scale would not be possible otherwise.
- The information is complete, all parts of software, documentation, test cases that are under version control are recorded.
- The way the version control system is used rarely changes, making data uniform over time.
- Even small projects generate large volumes of changes making it possible to detect even small effects statistically.

- The version control system is used as a standard part of the project, so the development project is unaffected by experimenter intrusion.

We believe that MRs are a very rich source of information about software development and that their analysis can evoke rewarding insights. Unfortunately drawing conclusions about characteristics such as effort, quality and interval is fraught with challenges. We describe some of these challenges in the following sections. Basic to all of them is that special care must always be taken to obtain information on how version control and change management are used in the project so as not to misinterpret the MR classifications or misunderstand the process used to create, make progress on, and record information about MRs.

3.3. Estimating change cost

While many attributes of changes are recorded by a version control system, the cost it takes to perform a change is impractical to collect. We consider cost to be paid developer time. Often we use the term effort instead, but not to refer to the actual effort or time it might take a programmer to complete a change, because the actual effort might depend on the programmer working unpaid or unrecorded overtime, on doing tasks that are not directly related to completing changes to software, such as attending meetings, learning, or, on occasions, simply being more or less productive than usual for other reasons.

While some change management systems, e.g., Sablime, include effort related fields such as estimated and actual effort, we found them to be rarely used and even when used to be extremely inaccurate. Interviews with developers in the latter cases indicates that they often put a constant number in such fields that does not depend on actual or estimated effort.

By making a few simple assumptions we can overcome this difficulty and determine, for example, what makes some changes cost more than others, as shown in [1], where the key idea is to tease out the extent of the contribution of various factors to the time a developer spends on a particular change. Our assumptions are as follows.

1. The attributes of software changes such as the identity of the individual making the change, size of the change, and types of tools used, determine the cost it takes to complete a change (more examples are in, e.g., [12]).

2. For each MR we have available the identity of the developer and the time when the MR was submitted, i.e., the time after which no work was done on that MR.

3. The work on the MR was done by the developer who submitted the MR and that work was started a short time before the submission of the MR: we used the limit of two to ten weeks before its completion for the start of an MR.

4. The developers on average spend approximately equal amounts of paid time every calendar time period (in a month, or in a week). The overtime was either not recorded or not paid in the organizations we studied so this assumption accurately reflects the cost of developer time to the organization.

5. It is possible to determine whether or not an MR is a repair or new feature. Both attributes are typically kept by a change management system, with the latter often being somewhat inaccurate. We use methods described in [11] to determine whether or not an MR is a repair.

Based on developer interviews in the commercial projects we studied these assumptions held except for infrequent instances when several developers contributed to the same MR.

3.3.1 Change effort estimation algorithm

Using the preceding assumptions we have previously developed an algorithm for estimating effort for changes [1, 7]. The algorithm consists of an initialization stage where a rough estimate of MR effort is obtained, followed by iterations. A regression model is used in each iteration of the algorithm to guide the refinement of the estimate. The code to perform the analysis is published in [7].

In the initialization step of the algorithm a table for each developer is completed (see Table 1). Table columns represent weeks, table rows represent MRs, and table cells represent the cost (developer paid time). The table is initialized as follows.

1. Put zeros in cells where no work on the MR was done (in weeks after the MR was completed and in weeks before the work has started on the MR). The MR start date is a parameter to the algorithm.

2. Put ones in column sums: we assume the monthly cost for each developer to be constant for that developer (see Table 1). The differences between developers are obtained in the model fitting stage.

3. Fill the remaining cells in each column by equally dividing the column sum among nonzero cells in that column.

4. Calculate the row sums by adding cell values in each row.

Each iteration consists of the following steps.

Table 1. An example table of effort-per-MR-per-week breakdown for one developer. The ?'s represent blank cells, whose values are initially unknown.

	W1	W2	W3	W4	W5	$\cdots$	Total
MR_1	0	?	?	?	0	$\cdots$	?
MR_2	?	?	?	0	0	$\cdots$	?
MR_3	0	0	?	?	?	$\cdots$	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
Total	1	1	1	1	1	$\cdots$	12

1. A regression is fitted using a number of attributes of each MR as predictors and the fitted coefficients are used to predict MR efforts.

2. The cell values in each row are rescaled to sum up to the MR efforts predicted by the regression model.

3. The cell values in each column are rescaled to sum up to the column sums (ones).

4. A new set of row sums is calculated from the cell values.

The iteration is repeated until convergence (we stopped when the sum of the absolute values of the changes of all the fitted regression coefficients becomes less than 0.01).

The next section describes the project model we used, or the way MRs relate to each other in a software project.

4. Project Model

The main goal of our project model is to predict the amount and the distribution over time of the maintenance effort. This may be done either at an advanced stage of the development, for example, at code complete date, or it could be done at an earlier stage based on the work breakdown structure.

To perform accurate prediction we need two basic properties from the model: the model has to relate the existing effort to future effort and it has to learn (incorporate information) from completed projects. We express these two properties as the following postulates:

1. Every unit of new feature effort generates μ units of repair effort with each unit of repair effort having independent delay (the interval between the submission of the new change and the submission of the repair) with mean value of $1/\lambda$.

   The underlying thesis is that addition of a new feature may make the software violate its specifications because the new feature is not implemented correctly or because the existing functionality is exercised in a novel way. Not all defects are likely to be detected immediately because they may manifest themselves at compilation, unit testing, integration, system testing, or in the field.

2. The organization has completed some projects that are similar to the project to be predicted, e.g., similar development team, process, and software functionality. This is to ensure that the parameters μ and λ, estimated from previous projects, provide useful information to the current project.

To use the model, the new feature effort is obtained by the effort estimation techniques described above when a significant amount of new features have been implemented, or from a work breakdown structure if the prediction is done earlier in the project.

As illustrated in Figure 3, the repair starts in parallel with new feature development. The figure shows new effort (solid line), repair effort (dashed line), and predicted repair effort (dotted line) for 11 releases. Notice that in some earlier releases a significant portion of new functionality is implemented before the repair effort peaks and the later releases had more iterative development model where new feature effort is distributed more evenly throughout the project.

In the following subsections, we will briefly describe a probability model to fit the observed data. The likelihood function is a statistical concept. It is defined as the theoretical probability of observing the data at hand, given the underlying model. Empirically, it is a function of the observed data and unknown model parameters (μ and λ). Maximizing the likelihood with respect to these unknown parameters will yield the so called Maximum Likelihood Estimate (MLE) of these parameters (denoted as $\hat{\mu}$ and $\hat{\lambda}$). This is a powerful statistical technique that allows us to generalize linear regression procedures to arbitrary statistical models.

4.1. Project likelihood

Assume that the project starts at time 0 and we observe defect and new feature effort up to time t. In most cases, the link between a repair change and its root cause cannot be observed. In other words, one does not know which of the previously implemented features caused the bug, only that one of them did. Borrowing terminology from standard probability theory, our postulate 1 in the simplest case implies that the repair effort generated by one unit of feature effort follows a nonhomogeneous Poisson process with intensity function $\mu\lambda e^{-\lambda t},\ t > 0$. Let $B_{[a,b]}$ denote the repair effort during the time interval $[a, b]$. The model implies that $B_{[a,b]}$ follows a Poisson distribution with mean

$\mu_{[a,b]} = \mu(e^{-\lambda a} - e^{-\lambda b})$. Future effort can be calculated by setting $b = \infty$, provided that one can estimate μ and λ.

Consider first the case when one feature is implemented at time 0. The data we observe can be described as n units of repair effort occurring at times $s_j, j = 1, \ldots, n$. Let $S^{(j)}$ be the order statistics of the repair times. Then the likelihood function, i.e., the joint probability distribution, of observing such data is

$$P(B_{[0,t]} = n, S^{(j)} = s_j,\ s_j \leq s_{j+1}, j = 1, \ldots, n) = e^{-\mu_{[0,t]}} \mu^n \prod_{j=1}^{n} \lambda e^{-\lambda s_j}. \quad (1)$$

The parameters μ and λ can be estimated numerically by maximizing the likelihood function.

Next, let us consider the more realistic case of observing N_{t_i} units of new feature effort at time t_i, and B_{s_j} units of repair effort at times s_j. Some slightly more tedious algebra shows that the negative log-likelihood is:

$$\sum_{t_i} \mu N_{t_i} \left(1 - e^{-\lambda(t-t_i)}\right) - B_{[0,t]} \log(\mu\lambda) - \sum_{s_j} B_{s_j} \log\left(\sum_{i:t_i<s_j} e^{-\lambda(s_j - t_i)}\right). \quad (2)$$

4.2. Limitation of the model

A problem occurs when applying the model to unfinished projects. Let us consider the simplest case of observing one unit of new feature effort at time 0 and one unit of repair effort at time s and the observation interval is $[0, t]$. When s is close to $t/2$, one can show that the MLEs are $\hat{\mu} \to +\infty$ and $\hat{\lambda} \to +0$! A similar argument can be made in more general settings. What this shows is that under fairly general assumptions it is impossible to predict what will happen in an isolated project until it reaches its late stages unless some additional information is available. We consider several ways to address this shortcoming of MLE:

1. Perform prediction in a project using estimates for one or more parameters obtained on completed projects.
2. Impose an informative prior distribution.
3. Use data from other completed projects.
4. Observe relationships between repairs and new features causing them.

The first approach is the simplest and works best when the completed projects are similar to the project under consideration. The second approach is to use a Bayesian estimation technique instead of an MLE. An informative prior would eliminate the impossible infinite values of the estimates. However, this would require an elicitation step so that an appropriate prior could be obtained from people intimately familiar with the project. While this may be the only option for a new project, it is possible to use data from completed projects in a more mature organization using, for example, a hierarchical model. Finally, modifying the problem tracking process to record the change causing the repair would provide the data needed to produce a usable estimate. Here we report the results based on the first approach.

5. Empirical Results

We fitted the model on a family of software projects each representing a release of a large real time high-availability software system. We considered 11 releases developed over more than 9 years with contributions of 494 developers via more than 27,000 MRs.

The older releases were used to fit the parameters of the project model and the estimated parameters were then used on a current release to obtain predictions of the future development effort.

The project used the Sablime system for MR tracking and a proprietary SCCS based system for version control. Certain parts of the project also use ClearCase for version control.

We consider a slightly simplified version of the development process, as follows. The developers are assigned a new feature or a defect to work on. In case of defects, they investigate the problem, make necessary changes and submit an MR for integration. In case of new features, additional tasks such as low level design and design review are performed prior to coding. After coding is complete the MR is submitted for integration by the developer. The code inspection is done afterward and any issues are resolved with additional MRs. If an MR is opened by a tester, it may take some time until someone is assigned to work on it and eventually starts working on it. In this case the MR open time may significantly precede the time when the work started. Often developers will find an issue to work on in the regular course of their activities. They may investigate the issue, complete the necessary changes in their private workspaces, and then open and immediately submit an MR for integration. In this case opening an MR does not precede the start of work. Consequently we could not use the MR open time as an accurate indicator of when the work was started.

The MR submit time, on the other hand, fairly precisely determines the completion of the work on an MR, because developers have no reason to keep an MR open after it is complete. Once an MR is submitted, it goes next to system integration build and then to system test. Such builds occur once a week or once every two weeks for the projects under investigation.

5.1. Estimation of change effort

As a baseline we assumed MRs to take at most two to ten weeks of calendar time counting backward from the date of submission. For each developer we created a table as in Table 1 with columns representing weeks and rows representing MRs. Initially the table is filled with zeros except for the four weeks up to the week when an MR was submitted. The nonzero cells are filled column by column, dividing unit weekly effort equally among the nonzero cells in the column. This corresponds to the initialization step of the algorithm described in [7, 1].

To refine these estimates we used several factors that we could measure of which we selected the ones that were shown to affect MR effort in previous studies, e.g., [1, 7], or we thought might influence MR effort:

- Developer. Login of the individual who worked on and submitted the MR.
- MR type. We classified MRs into new development (referred to as new features and enhancements in the considered projects), repair, and field problems. The last class corresponded to repairs done in response to a customer problem reported on released software.
- MR status. We discriminated among MRs that had status "nochange" indicating that no change to the code was done as a result of the MR, versus the rest.
- MR size. Logarithm of the number of lines added by the MR.
- MR size/complexity. Logarithm of the number of deltas constituting the MR.

The estimated regression coefficients and their 95% confidence intervals are presented in Table 2. The regression equation by which the coefficients affect effort is as follows:

$$E(\text{Effort}) = (\text{delta} + 1)^{\alpha_1} \times (\text{lines} + 1)^{\alpha_2} \times e^{\alpha_{nochange} I(nochange) + \alpha_{new} I(new) + \alpha_{field} I(field)} \times \prod_{i:\{\text{developers}\}} \beta_i I(developer_i). \quad (3)$$

where I is an indicator function. The non-field repair MR resulting in a code change is the baseline for the coefficients $\alpha_{nochange}$, α_{new}, and α_{field}. For example, a new feature MRs take $e^{0.39} \approx 1.5$ times the effort of comparable repair MRs.

Table 2 shows that the effects are in the expected direction, with field problems and large MRs requiring more effort. New feature MRs are more difficult in the considered projects than repair MRs. This is in contrast to the finding in [1], where defect repairs required more effort. The rest of the predictors are not significant in predicting effort. The difference can be explained by the fact that [1] considers a different, even larger, software product. Furthermore, in [1] the algorithm used MR open time as an indication of when the work on an MR was started. Because new feature MRs are more likely to be opened after most of the work on the feature has been completed, e.g., low level design and prototyping, and repair MRs are more likely to be opened before the work has even started, this may have biased the estimates in [1] to make repair MRs appear more difficult than new feature MRs.

Table 2. Results from fitting effort model.

Coefficient	Estimate	p-val	95% CI
$\alpha_{nochange}$	-0.11	0.244	$[-0.30, 0.08]$
α_{new}	0.39	0.000	$[0.23, 0.55]$
α_{field}	0.69	0.000	$[0.56, 0.82]$
delta: α_1	-0.11	0.069	$[-0.22, 0.01]$
lines: α_2	0.15	0.000	$[0.10, 0.20]$

5.2. Fitting software project model

We can now use the calculated MR effort to model the project. We chose to discretize project time into work weeks because the effort estimation used weeks and because it simplified calculations. The total project effort spent during the week t_i was divided into three types: new effort N_{t_i} that was spent on MRs submitted that week that were classified as "new", repair effort B_{t_i} corresponding to repair MRs, and field effort F_{t_i} corresponding to field repair MRs.

Figure 3 illustrates the new and repair effort time series for the considered projects. The figure also shows a third curve that represents prediction of the repair effort as described in Section 5.3.

The maximum likelihood estimator for the completed projects in Table 3 gives 19 weeks delay for repair and 41 weeks of delay for field repairs. The ratio of total repair before field release to new feature effort in a project is 1.4 and for the total field problem repair effort to new feature effort is 1.7. Please note, that these are not efforts for individual MRs, but rather total efforts spent in the project on new functionality, repair, and repair of field problems.

Table 3. MLE estimates for the delay and repair to new feature effort ratio.

Type	Delay	Delay CI	Ratio	Ratio CI
Regular Repair	19	$[18, 21]$	1.36	$[1.2, 1.5]$
Field problem	41	$[33, 48]$	1.7	$[1.4, 2]$

Here we observe fairly small 95% confidence intervals

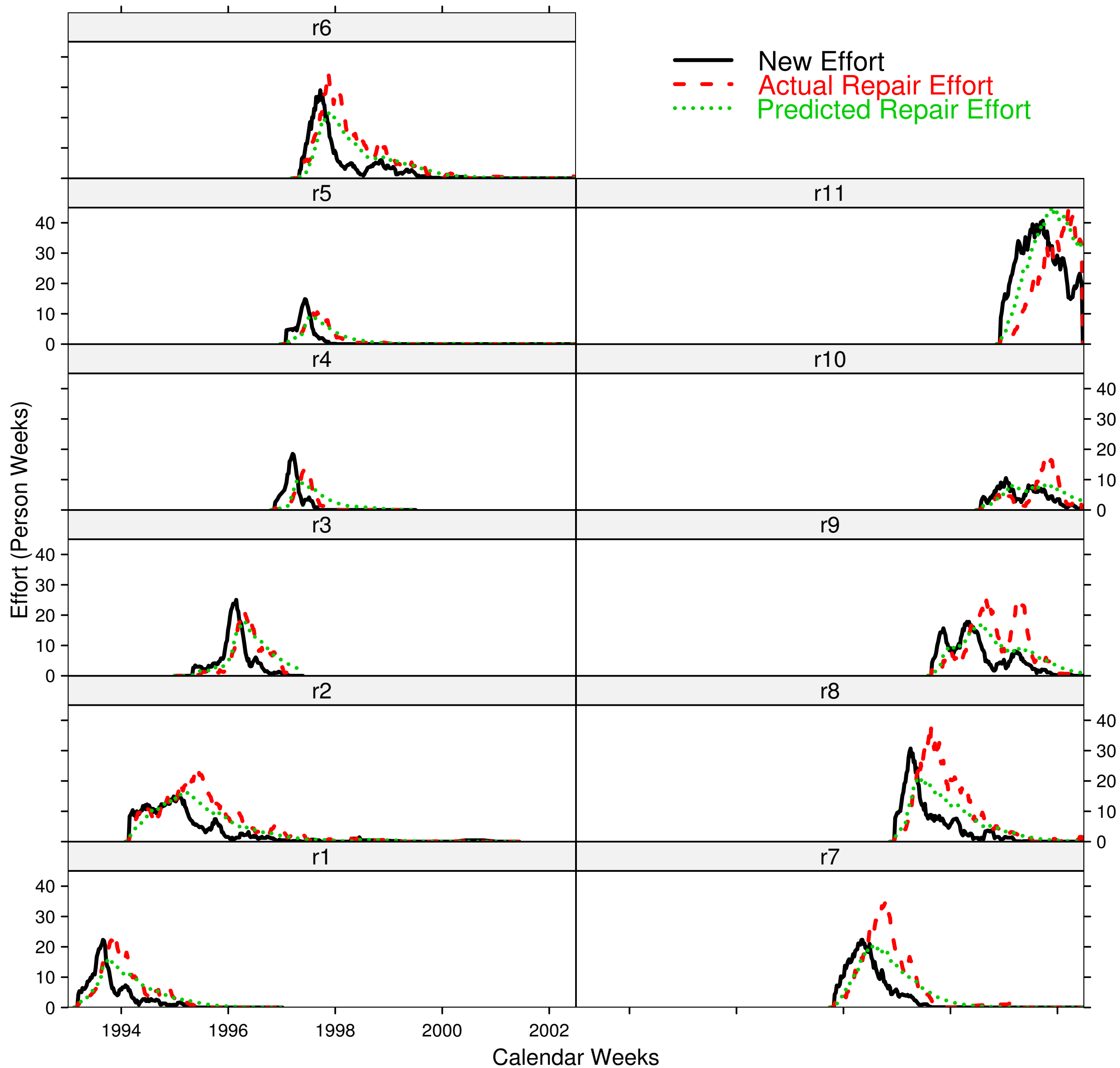

Figure 3. Effort trend broken by the type of change for several releases.

in terms of delay and in terms of the effort ratio. This gives us confidence that these estimates would accurately predict the repair effort and timing of the future projects.

Note, that total effort to implement new features represents only one quarter of the total effort in the considered software projects with repair of problems found in the field constituting almost half of the total effort. This is partially caused by the fact that the field repair MR is about twice as difficult ($e^{0.69}$) than a comparable non-field repair MR. The results for these projects confirm and make more precise the oft-heard statement that the later one finds a problem the more difficult it is to fix.

5.3. Prediction of repair effort in a project

Armed with the delay and ratio estimates from historic projects we can proceed to prediction of the repair effort. Our key assumption is that we approximately know the new feature implementation schedule and resources devoted to accomplish it. Such a plan is typically available at the end of the project planning stage before the development starts. Of course, the actual schedule might differ from the planned schedule because of a number factors, including the unplanned drain of resources to repair already released projects.

We applied our method halfway through the ongoing project in November 2001, for the project that was started

in the beginning of the year and had a major release in May of the subsequent year. Consequently, we had information (new and repair MRs) of the already completed part of the project. In Figure 3 we illustrate the predicted repair effort using project model coefficients estimated in the previous section: repair delay of $1/\hat{\lambda} = 19$ weeks and repair to new ratio of $\hat{\mu} = 1.36$. According to the project model, the predicted intensity of repair effort at time t is simply an integral $\int_0^t N(s)\hat{\mu}\hat{\lambda}e^{-\hat{\lambda}s}ds$. Given our discretization of time into weeks the predicted repair effort for week t_k becomes the sum $\sum_{t_j<t_k} N_{t_j}\hat{\mu}(e^{-\hat{\lambda}t_j} - e^{-\hat{\lambda}(t_j+1)})$.

Such prediction does not take into account the fact that each project has a slightly different relationship between repairs and the new features. Consequently, if the prediction is done at the advanced stages of the project, as was the case in our study, valuable information about the project is not used. To address the problem we need to estimate project specific parameters μ and λ. However, because of reasons described in Section 4.2, it is impossible to estimate both parameters using MLE. We tried the options described in 4.2 and got similar results, but because of space considerations we report results from the simplest first option where we find a new estimate just for λ, keeping $\hat{\mu}$ estimated from previous projects. The estimate for $1/\lambda$ done in November 2001 was 48 weeks, much larger than for other projects. Figure 1 illustrates the performance of the repair schedule prediction for release r11. The first predictor uses $\hat{\mu}$, $\hat{\lambda}$ obtained from other projects and the second uses project specific estimate for the delay of 48 weeks.

Clearly, both predictors follow the shape of the repair activity, with the first predictor having an overly optimistic view about when the repairs will be completed and the second predictor possibly having an overly pessimistic view (the predicted repair curve continues well beyond July, 2002, the time of this writing). Only time will tell if the second predictor is in fact, overly pessimistic.

The overly optimistic performance of the first predictor that relies on the historic projects is not difficult to explain. The project under study was extremely ambitious, larger than any project undertaken before, involving new types of functionality. At the same time, the project schedule was highly optimistic for the amount of work that needs to be done, leading to various process experiments where work on numerous new features simultaneously was initiated and the job of system integration and testing was left for later stages, partly because there were not sufficient resources to do new development and sufficient testing at the same time. As is clear from the illustration, the strategy had a mixed success, with repair effort being delayed as compared to other releases.

6. Validation

Because the defects are discovered at the compilation, unit test, integration, system test, and field phases, it may make sense to consider time until repair to be a mixture distribution, each representing the phase at which the defect is discovered.

We chose to separate defects into field and non-field repair because it was possible to perform that separation and because we expected the time until discovery of field defect to be much longer than the time until discovery of other defects. The results in Section 5 indicate that, indeed, the field problems take twice as long to manifest themselves as regular defects.

At first glance it may seem that the assumption that only new feature changes generate repair changes is too restrictive. After all, fixes sometimes cause problems too. From the modeling perspective there are two ways to deal with this issue. The defects that cause other defects were caused by the new feature (it may even be the original code), so such a non-first generation repair could be attributed to the new feature change. This, of course, would increase the overall delay parameter and may require using a distribution for the delay that has a heavier tail than the Exponential distribution that we used here. We have tried using Weibull, Pareto, and Gamma distributions in the project model, but the results do not bring any surprises or significant lessons to be worth reporting. In the second approach, a complete likelihood may be written for the model where both new features and repair generate repair. We have done so, but it is computationally more complex and we have yet to apply it to real data.

We tried to incorporate all the uncertainties that we are aware of into our project model so the variability of the estimates would reflect the known uncertainties. More specifically, while estimating MR effort we considered several possible values for how long before MR submit time the work on MR may start. We considered 2, 4, 6, and 10 weeks and obtained effort estimates for each case. The results in Tables 2 involve bootstrap (a statistical term referring to calculating estimates based on multiple subsamples of the data) by sampling this parameter and by selecting 20 non-overlapping subsets of developers.

Table 3 incorporates the MR open time uncertainty by showing the mean and the confidence interval of the estimates taken over the four possible MR open times and over all 10 releases previous to R11.

Of course, from the project planning perspective the ultimate validity is the accuracy with which the repair effort and schedule could be predicted as illustrated in Figure 1.

7. Project Model Scenario

The utility of the presented model in other projects can be shown if the model is used in project planning or project understanding. In this section we outline the steps needed to apply the model in a software project. There are four basic stages: change data extraction, change data validation, and change and project modeling. In the data extraction stage access to the project systems is obtained and raw change data is extracted. In case of home-grown tools, it may be necessary to interview a person responsible for tool support to understand the structure and functionality of such systems.

The most important is data validation stage which starts from cleaning the raw change data to eliminate administrative, automatic, post-preprocessor, and other computer-generated changes. This leads to data described in Figure 2. The quality of attributes is then assessed and un- or auto-populated attributes and remaining system generated artifacts are eliminated. When engaging a new project it is important to interview a sample of developers and testers. The interview involves review of recent changes done by the interviewee to to illustrate the actual development process and to understand/validate the meaning various attribute values.

The change data is then augmented by modeling change effort [1] as described above, and, if needed, change purpose [11] or risk [12]. This stage also requires validation described in the references. It is worth pointing out, that at this point a number of important things have been learned about the software organization, including relative efforts for repair and new feature changes, effort per average change, change and effort trends over time and for different projects.

Once the relevant information on software changes is obtained, a project model is fitted for historic releases as described in Section 5.2. While we divided repair changes into post-release and pre-release repair, other classification might be desirable in other organizations. For example, classifying changes due to unit, feature, integration, and system test. For each type of repair and each project, the fitted values represent relative amount of repair effort and an average time until repair. Investigating these estimates for different projects it is important to note the variability and any patterns where the size or other characteristics of the release appear to affect the amount or delay of the repair effort.

Finally, the model may be used to predict repairs and answer other project planning questions for the upcoming project as described in Section 5.3.

8. Related Work

Our work closely relates mainly to two areas in software engineering: cost and schedule estimation and risk assessment. The software cost estimation may be roughly organized into expert and algorithmic techniques to estimate software cost and schedule.

The expert based techniques are typically best suited for projects that are not too different from projects completed in the past and where the estimator has extensive experience of estimation with these past projects (a good review of expert estimation techniques may be found in [9]). The main drawback is the subjective and non-transparent nature of the estimation process that makes it harder to justify the estimates. Often, it is also difficult to find estimators with the appropriate experience in the application and the environment in which it is developed. Our method is largely complementary to expert estimation, for example, expert estimates on the new feature effort and schedule can be fed into our model to predict the repair (and total) effort and schedule before the development has started. To help experts come up with more precise estimates they can inspect actual new feature and repair effort schedules for the past projects as shown in Figure 3.

Algorithmic techniques such as COCOMO [2, 3] may be used if the key predictors, such as size of the project, can be reliably estimated in advance and calibrated with past projects. The main drawback is that the size of the project (an input to the algorithm) may be more difficult to estimate than the cost (the output of the algorithm). While COCOMO is typically used to estimate the entire cost of the project, it may be possible to calibrate the regression used in COCOMO to estimate new feature and repair effort separately. As with expert-based techniques the estimates for new feature effort could be fed into our model and the historic MR effort data could be used to calibrate the COCOMO.

The risk assessment literature covers a number of issues, but the part most related to our work predicts the number of defects remaining in software during testing [14, 5, 6, 16]. There are two key differences with our work: we do not predict defects based on observed defect counts, but rather predict defect effort schedule based on observed new feature changes. The second difference is that testing models assume that the software does not change during testing, which is clearly not true as indicated by Figure 3. Rather, our model assumes that new features are continuously added to the software (albeit at a time varying rate) and generate defects and need for defect repair effort later on.

9. Discussion

We find it somewhat surprising that the relationship between new features and repair in a software project could be reasonably described using just two parameters: the proportion of repair effort and delay until repair. Encouragingly,

these parameters vary little across projects in a single organization making them suitable for the planning of upcoming projects.

While our focus was fairly narrow: to predict repair effort and its distribution over time given the new feature effort, the project model can be applied in more general settings. For example, it could predict the number of defects based on the number and properties of new features as in [12], or predict the defects in a software module as in [8]. Furthermore a classical testing problem [5] of when to stop testing by predicting the number of defects remaining in the code base could be solved using the same model.

Since the model uses data that is available in virtually any large software project it is an appealing model for commercial software settings where all process overhead, including time-consuming data collection, are treated with skepticism.

In addition to providing a framework for answering a number of important project planning questions, the project model may help gain better insights into completed projects and may help a software organization better understand its strengths and shortcomings. One significant benefit that we expected and observed, was better understanding of a software projects constraints by people who are not directly involved in development, yet have to take important planning and resource allocation decisions.

As an example, the model applied to the projects we studied confirmed and quantified the customary wisdom that the later a defect is found the harder it is to correct (and the longer it takes to correct it). Such quantification makes it easier to justify expending resources earlier in the project to detect and correct defects before they reach the field. We leave it to future research to quantify the efficiency of various detection techniques, such as code inspections, so that one could estimate just how much effort should be expended on such a technique to make it worthwhile.

Finally, much of our ability to model projects was a result of making relatively few and minimal assumptions about how the work in a software development project proceeds based on our observations of the development process. Our observations and subsequent analysis focused on what changes were made and how they were made, i.e., how people worked to change the code, rather than focusing on properties of the code. Although we believe that analyzing code properties has its place, we also believe that analyzing changes is a rich source of insight that is usually not given proper attention.

Acknowledgments

We would like to thank all the people in Avaya, Lucent, and AT&T who provided information directly (via interviews) or indirectly (by working on the products under study.) In particular we thank S. Brown, M. Flores, D. Frost, R. Hackbarth, N. Harrison, M. Hazerodt, J. Maranzano, O. Mascarenhas, E. Moritz, S. Muchow, J. Palframan, J. Payseur, D. Sokoler, D. Turgeon and others for providing insight on development process, development tasks, project management and other aspects of the studied software projects.

References

[1] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, July 2002.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[3] B. W. Boehm, B. Clark, E. Horowitz, and et al. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1(1):1–24, November 1995.

[4] P. Cedeqvist and et al. *CVS Manual*. May be fond on: `http://www.cvshome.org/CVS/`.

[5] S. R. Dalal and C. L. Mallows. When should one stop testing software? *Journal of American Statist. Assoc*, 83:872–879, 1988.

[6] A. L. Goel. Software reliability models: Assumptions, limitations and applicability. *IEEE Trans. Software Engineering*, SE-11(12), 1985.

[7] T. Graves and A. Mockus. Identifying productivity drivers by modeling work units using partial data. *Technometrics*, 43(2):168–179, May 2001.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[9] M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 2002. submitted.

[10] A. K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[11] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, pages 120–130, San Jose, California, October 11-14 2000.

[12] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April–June 2000.

[13] A. Mockus and D. M. Weiss. Globalization by chunking: a quantitative approach. *IEEE Software*, 18(2):30–37, March 2001.

[14] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGrawHill, New York, 1987.

[15] M. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, 1(4):364–370, 1975.

[16] N. D. Singpurwalla and S. P. Wilson. *Statistical Methods in software Engineering: Reliability and Risk*. Springer Series in Statistics. Springer-Verlag, New York, 1999.