Using Version Control Data to Evaluate the Impact of Software Tools

David Atkins,[*] Thomas Ball,[*] Todd Graves[*+] and Audris Mockus[*]

[*]Software Production Research Dept.
Bell Laboratories, Lucent Technologies

[+]National Institute of Statistical Sciences

{dla,tball,graves,audris}@research.bell-labs.com

ABSTRACT

Software tools can improve the quality and maintainability of software, but are expensive to acquire, deploy and maintain, especially in large organizations. We explore how to quantify the effects of a software tool once it has been deployed in a development environment. We present a simple methodology for tool evaluation that correlates tool usage statistics with estimates of developer effort, as derived from a project's change history (version control system).

Our work complements controlled experiments on software tools, which usually take place outside the industrial setting, and tool assessment studies that predict the impact of software tools before deployment. Our analysis is inexpensive, non-intrusive and can be applied to an entire software project in its actual setting. A key part of our analysis is how to control confounding variables such as developer work-style and experience in order accurately to quantify the impact of a tool on developer effort.

We demonstrate our method in a case study of a software tool called VE, a version-sensitive editor used in Bell Labs. VE aids software developers in coping with the rampant use of preprocessor directives (such as #if/#endif) in C source files. Our analysis found that developers were approximately 36% more productive when using VE than when using standard text editors.

Keywords

software tools, version control system, effort analysis

1 Introduction

While software tools have the potential greatly to improve the quality and maintainability of software, acquiring, deploying and maintaining a tool in a large organization can be an expensive proposition. We explore how to quantify the effects of existing software tools in ongoing large-scale software projects, presenting a simple methodology that correlates tool usage statistics with effort estimates based on analysis of the change history of a software project. Our work complements controlled experiments on software tools, which usually take place outside the industrial setting, and tool assessment studies that predict the impact of a software tool before it is deployed.

Our work is based on two observations. The first observation is that a major effect of a software tool, be it a documentation tool, source code editor, code browser, slicer, debugger, or memory-leak detector, is to help a developer determine how to modify a software entity or to directly to aid the developer in making modifications. The second observation is that the change history of a software entity (i.e., the version control data about the modifications to the entity) can be used to estimate the amount of effort a developer expended on a particular modification or set of modifications. These observations lead to a simple process for assessing the impact of a software tool:

1. Record the tools a developer uses in the course of software development and the software entities to which they were applied.
2. Correlate the monitoring information recorded in step 1 to the modifications to software entities that are recorded by the version control system.
3. Using the data from the previous two steps, analyze "similar" developers and modifications[1] to estimate how the use/non-use of the tool affected developer effort.

As we will see, this process is automatic, inexpensive, non-intrusive, and applicable to arbitrary software projects using version control systems. Furthermore, it can be applied to an entire software project in its actual setting.

[1]Section 5 will qualify and quantify the notions of "similar" developers and modifications.

To appear, 1999 International Conference on Software Engineering.

We apply our process to a real-world example from Lucent Technologies. We present a case study of a software tool that provides an elegant solution to the problem of rampant use of preprocessor directives (such as `#if/#endif`) in C source files. These directives typically are used to create many different variants, or versions, from a single source file. A developer editing such files must be careful to make changes to the appropriate version, so as not to interfere with other versions. [25] The solution to this problem is a version-sensitive editor (VE) that hides the preprocessing directives from a developer. VE allows a developer to edit a particular version of the source file (i.e., a view of the underlying ASCII file in which certain preprocessing directives have been "compiled" away). As the user edits this view of the source code, VE translates editing operations on the view back into the underlying source file.

Our hypothesis is that the VE tool reduces the effort needed to make changes involving pre-processor directives. We test this hypothesis via a quantitative analysis of developer effort based on the change history of a very large software product in which both VE and other text editors were used. For each change made to the software, we were able to determine whether or not VE was used to make the change. By combining this information with the developer effort analysis, we found that developers who used VE were approximately 36% more productive than when using standard text editors (when changing files containing preprocessor directives).

Our case study points the way to a general methodology for evaluating software tools in the industrial software development environment, as outlined above. Through our case study, we illustrate a number of problems that must be solved to arrive at an accurate estimate of how software tools impact developer effort. Primarily, these are problems of how to control for key sources of variation such as:

- *Developer work-style and experience*;
- *Size of changes to software*;
- *Type of changes (new feature, bug fix, code cleanup, code inspection).*

Our work is complementary to the analysis of tools in controlled settings and software tool assessment. Controlled experiments on tool use can yield valuable insights about the utility of a tool on small scale examples; our work seeks to address the ongoing impact of a tool in the industrial developer population at large. Software tool assessment compares various tools to one another and attempts to predict the impact of a tool on a project before deployment.

The paper is organized as follows. Section 2 provides background on version control systems. Section 3 describes the version editor (VE) tool, how it addresses the problem of preprocessor directives, and how we were able to monitor the usage of VE over more than a decade of use in a large software project. Section 4 summarizes our methodology and algorithm for analyzing version control data in order to estimate the effort necessary for developers to make changes. Section 5 presents the results of applying this algorithm to the version control data from a large software system in which VE and other text editors were used. Section 6 describes a general framework for repeating this experiment in other settings. Section 7 discusses related work.

2 Background

The case study here revolves around a commercially successful multi-million line software product (a large telephone switching system) developed over two decades by more than 5,000 developers.

The extended change management system (ECMS) [16], layered on top of the source code control system (SCCS) [22], was used to manage the source code.

We present a simplified description of the data collected by SCCS and ECMS that are relevant to our study. ECMS, like most version control systems, operates over a set of files containing the text lines of source code. 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 SCCS, which compares an older version of a file with the current version.

ECMS records the following attributes for each change: the file with which it is associated; the date and time the change was "checked in"; and the name and login of the developer who made it. Additionally, the SCCS database records each delta as a tuple including the actual source code that was changed (lines deleted and lines added), login of the developer, MR number (see below), and the date and time of change.

In order to make a change to a software system, a developer may have to modify many files. ECMS groups atomic changes to the source code recorded by SCCS (over potentially many files) into logical changes referred to as Maintenance Requests (MRs). There is one developer per MR. An MR may have an English abstract associated with it that the developer provides, describing the purpose of the change. The open time of the MR is recorded in ECMS. We use time of the last delta of an MR as the MR close time. We use keyword spotting of the MR abstract to infer the purpose of a change [17]. Each MR is classified to be the result of new feature development (NEW), bug fixing (BUG), code restructuring/cleanup (CLEANUP), or code inspection

fixes (INSPECT) based on the presence of appropriate words in the one line English text MR abstract recorded by ECMS.

3 VE: A Version-sensitive Editor

The software product in our case study requires the concurrent development and maintenance of many sequential versions as well as two main variants for domestic and international configurations of the product. From a version management point of view, source code may be common to as many as two dozen distinct releases of the code. Some of these releases correspond to deployed products for which only maintenance changes are made, while others correspond to versions under active development.

The software releases form a version hierarchy with two main variants and chronological release sequences within each of these. Several constraints on the project management are reflected in the way source changes are made to preserve this hierarchy. First, it is imperative that the new development or maintenance changes made for one software release not impact the previous release in the sequence or any release in the other main variant. Second, it is important that as much commonality of code be preserved as possible: changes made in an earlier release should automatically appear in the later releases in that sequence. In the examples that follow, the two main variant lines are designated as 'A' and 'B', and the sequential releases within each main line are designated by ascending numbers, e.g., 1A, 2A, 1B, 2B, and so on. To achieve the second objective, most of the source files are shared among the releases, with release specific differences delineated as described in the following paragraphs.

The industrial source code management technology of the early 1980's did not have good support for branching. That is, there were no tools for maintaining source that was mostly common to many releases but contained some release specific changes, and no tools for automatically merging separate changes to a common code base. To address the multiple release requirements of the project under study, a specialized directive #version was used to allow for release specific variations in the code, as shown in Figure 1. The #version construct permits a single source file to be extracted to produce a different version for each software release. We can think of this construct as a C preprocessor #if where only one Boolean symbol is used for control, the symbol may be negated, and the symbol comes from a restricted set that contains one symbol for each software release. Various tools are used to verify the consistent use of these constructs according to a release hierarchy maintained by the system. For example, the tools guarantee that a change checked in for 5A will not affect the source extraction for 4A or earlier or any of the 'B' releases. Tools

```
   ...
   if (!PreCheckRoute(route))
      return FAIL;
#version (4A)
   dest = GetDest(route);
   if (dest.port == 0) {
      return(RouteLocal(route));
   }
#endversion (4A)
   DoRoute(route);
   ...
```

```
   ...
   if (!PreCheckRoute(route))
      return FAIL;
#version (4A)
   dest = GetDest(route);
#version (! 5A)
   if (dest.port == 0) {
#endversion (!5A)
#version (5A)
   if (dest.port == 0 || dest.module == 0) {
#endversion (5A)
      return(RouteLocal(route));
   }
#endversion (4A)
   DoRoute(route);
   ...
```

Figure 1: Before and after a Release5A change. Emboldened lines are the code added by the programmer.

are also provided to perform the extraction of the source code for building each software release, again according to the version hierarchy. For example, extraction for release 4A implies that all version directives for 4A, 3A, 2A, and 1A are true and all other version directives are false.

When a developer introduces new code for a release, the new code must be bracketed by a #version construct for the specific release for which the change is targeted. When a developer changes existing code for a release, the existing code must be logically removed with a #version construct using the negation of the target release, and the change introduced with a #version for the target release. Figure 1 shows how these constructs are used to change the expression in an **if-then** statement for Release 5A. The original **if-then** statement was code inserted for Release 4A.

As the example shows, even a one line change to the code requires the developer to add five lines to the file (four control lines and the changed code line). The developer must bracket the original line with the negated

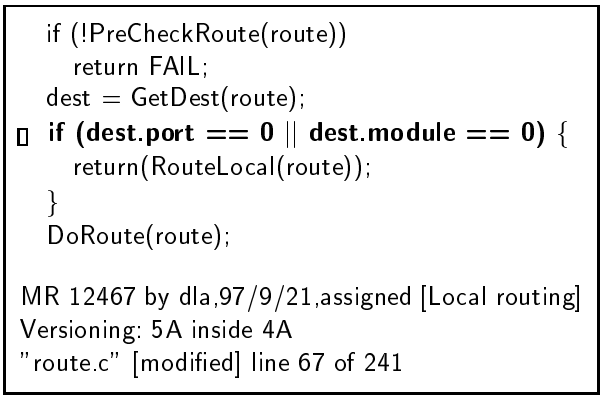

Figure 2: Release 5A view in VE with change in bold

#version control to omit it for release 5A. Then the developer makes a copy of the line and brackets it within #version controls for release 5A. Finally, the change is made to the copied line. In addition to all these actions for just a logical one line code change, the version control lines also make the source file more difficult to read and understand. For the project being studied, several dozen distinct releases have accumulated; some core source files may contain #version directives for most of these releases. In worst case files, only 10% of the lines of the file are the extractable source code for a release, with 50% of the lines being #version/#endversion lines and the other 40% being source that extracts for other releases.

To make this situation more manageable for the developer, a version-sensitive editor (VE) was made available [7, 20, 2]. This tool allows the developer to edit in a view that shows only the code that will be extracted for the release being changed. The tool also performs the automatic insertion of any necessary control lines. For example, the insertion of a new line for release 5A in an area that does not have any release 5A code will automatically produce the required #version around the line. Likewise, a change to a line will automatically produce the #version for the negation of 5A which will exclude the existing line for 5A, and insert the changed line with #version to include the change for 5A.

The developer's view is of normal editing in the extracted code; VE manages the changes to the #version constructs according to the described constraints. Figure 2 shows the view presented by VE for the file from Figure 1. In VE, the developer only has to use standard editing commands to effect the change to the **if-then** statement, and VE inserts the required #version directives (behind the scenes). VE behaves like either **vi** or **emacs**, the two standard editors used by most of the developers in the project. In fact, the appearance to the developer is that of using the standard editor with the extended behavior of dealing with #version lines automatically.

For this study, a noteworthy aspect of VE is that it leaves a signature on all of the #version/#endversion control lines that it generates. This signature consists of trailing white space (a combination of space and tab characters) that uniquely distinguishes the control line from any control line generated for any other change.[2] This was done to avoid unwanted dependencies caused by SCCS's use of the Unix diff. Source files can contain many identical #version/#endversion, and this similarity can in some cases cause SCCS to store a change as if it affected #version lines that the developer did not touch. VE essentially mimics an observed manual practice done to avoid this type of dependency. However, VE produces the trailing white space on *every* #version line it generates with an algorithm that uniquely identifies the lines as produced by VE. Since the use of VE is optional in the project, this "feature" of VE allows us to distinguish when VE was used to make a change involving #version lines from when the change was made using an ordinary editor.

Figure 3 shows the history of VE usage in the considered project, which consists of approximately 600,000 MRs. The three lines show the percentage of MRs that were done with VE (V: MRs such that all deltas of the MR contained #version lines with the VE signature), without VE (H: MRs such that some delta of the MR contained a #version line without the VE signature), and without #version lines (N: MRs such that no delta in the MR contained a #version line). The usage of VE increased dramatically over time.

4 Developer Effort Estimation

Since VE leaves a visible signature in the version history, all the necessary data are in place for measuring how helpful VE can be to developers. We hypothesize that when making changes involving #version lines, developers are more effective when using VE than when using standard text editors. In this section we describe a general methodology, introduced in [10], for measuring the influence of various factors on the effort required to make a change, using the change history of a version control system and periodic time sheet data. In Section 5, we apply this methodology to the problem of measuring the effect of the VE tool.

In principle, if measurements of effort for each change

[2] In fact, the trailing spaces and tabs encode the current delta number. As a result, even if developers copy VE-generated #version lines using an ordinary text editor, we can determine that this was a hand change with high probability (because the delta number of the signature will most likely disagree with the current delta number of the underlying SCCS file).

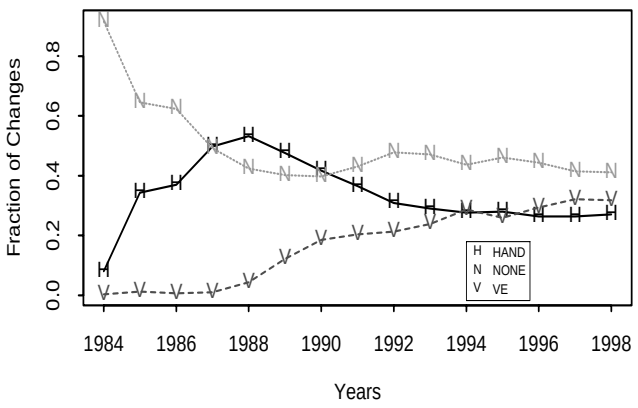

Figure 3: VE usage over time.

completed by developers were available to us, we could fit a regression model such as

$$E(\text{effort}) = \alpha_{\text{DEV}} \times \beta_{\text{TYPE}} \times \text{Size}^{\gamma} \times \theta_{\text{TOOL}} \quad (1)$$

in order to obtain estimates of the effects on effort of the following variables:

- DEV: developer identity;
- TYPE: type of change, which ranges over the values NEW, BUG, CLEANUP, INSPECT;
- Size: size of change, which is the number of deltas in an MR;
- TOOL: use or non-use of VE, which ranges over the values VE, HAND, NONE (NONE means the change did not contain any #version lines).

For a discussion of which variables it is important to include in the model, see [11].

Unfortunately change management systems do not record measurements of developer effort, so our algorithm makes use of monthly time sheet data instead. Table 1 illustrates, for a single developer, the data we normally have available to us. Rows in the table correspond to changes completed by the developer, and columns to months, so that each cell in the table is the amount of effort the developer devoted to a particular change in a given month. Monthly time sheet data tell us the sums of the entries in each column: how much total effort a developer expended in a month. We also know which changes a developer worked on during each month, and a developer's total effort needs to be divided across these changes.

	Jan	Feb	Mar	Apr	Total
Effort for MR A	?	?	?	?	??
Effort for MR B	0	?	?	0	??
Effort for MR C	0	0	?	?	??
reported effort	0.8	1.2	0.9	0.8	3.7

Table 1: Data available in effort estimation problem.

The row sums, if we knew them, would be effort measurements for each change, and we could use regression to relate these measurements to quantities such as the size of the change or whether the tool was used. The idea behind the algorithm is to begin with a guess at the change efforts and alternately use regression models and the time sheet data to refine our initial guess. In the process we will refine our understanding of the factors that affect change effort through the changing coefficients in the regression models. To construct an initial guess, we divide up each known monthly effort equally across all changes open in that month, and then repeat the following four steps until convergence:

1. Compute row sums to obtain estimates of total MR efforts, for each developer.
2. Fit a regression model of imputed MR effort on the factors that predict MR effort. We prefer to use generalized linear models [15] of the form of Equation (1).
3. Rescale the rows in the imputed monthly MR effort table so that the new row sums are equal to the regression's fitted values. Do this for each developer.
4. For each developer, rescale the columns of the table so that the column sums are equal to the observed monthly efforts.

The algorithm has converged in every application we have tried, and in fact there appears to be an underlying descent algorithm; see [11]. Ten or fewer iterations are generally sufficient for establishing the regression coefficients to three significant figures. After convergence, we report the coefficients in the final regression model.

Since the regression model is necessary for improving our estimates of change effort, it is necessary to make sure that the model includes quantities which are known to be closely related to change effort. We have found that the models should include coefficients which depend on the developer, since variations in developer productivity are often quite large [3, 8]. The model should also include a measure of the size of a change, such as the number of lines changed or the number of atomic changes making up the change. Whether the change is

a bug fix, new feature development, cleanup effort, or inspection rework, is also important.

We have found that because reported monthly efforts are similar across months, one can replace these data using the assumption that each developer contributes one unit of effort each month, without changing the results substantially.

An important component of the inference methodology is assessing how certain one can be about the values estimated for the coefficients in the final regression model. As discussed in [10], we use the "jackknife" method, which consists of removing one developer from the list we used, running the algorithm again, repeating once for each developer, and observing how much the coefficients change depending on which developer is omitted.

5 Effectiveness of the Version-Editor Tool

This section tests our hypothesis that the VE tool reduces the effort needed to make changes involving #version lines. Our analysis proceeds in three steps:

1. Tag each delta and MR with VE signature information;
2. Select a balanced set of developers;
3. Estimate the effect of the VE tool using the effort estimation algorithm of the previous section.

At the end of the section we summarize measures taken to ensure the validity of the results.

Extraction of VE signature for each delta

As described in Section 3, VE leaves a signature in SCCS files because of the trailing white space it inserts after the #version/#endversion lines. We wrote a program that processed all 27 gigabytes of SCCS records for the software project under consideration and identified three attributes for each delta:

1. number of #version lines;
2. number of #version lines with VE signature;
3. number of #version lines without VE signature.

This information was used to identify the deltas where the usage of VE was not likely to have impact (i.e., those deltas that contain no #version/#endversion lines), and where the usage should have an impact (presence of #version/#endversion lines).

As defined in Section 2, an MR typically consists of several deltas. It is possible that some of the deltas in one MR have a VE signature and others do not. This does not happen frequently: only 1.8% of the MRs had this property in the entire dataset of 600,000 MRs and in the analyzed sample of 3,400 MRs (we selected this sample of MRs by choosing a subset of developers as described below. We marked such changes for analysis purposes as made by hand, since according to our null hypothesis (VE does not reduce developer effort for changes involving #version lines) such marking should not have any impact. If, however, VE reduces developer effort, then such marking would only make it more difficult for the VE effect to show up as statistically significant.

Developer selection

The variability in project size, developer capability and experience are the largest sources of variability in software development (see, for example, [3, 8]). The effects of tools and process are often smaller by an order of magnitude. To obtain the sharpest results on the effect of a given tool in the presence of developer variability, it is important to have observations of the same developer changing files both using the tool and performing the work without the aid of the tool.

We focused on developers who made substantial numbers of changes requiring modifications of #version/#endversion lines, both with and without the VE tool. Also it is preferable to consider developers that had similar work profiles (i.e., made similar numbers of changes). Given the considerable size of the version history data available, both tasks were easy: we selected developers who made between 300 and 500 MRs in the six year period between 1990 and 1995 and had similar numbers (more than 40) of MRs done with and without VE.

Effort drivers

We fitted the model (Equation (1) from Section 4), estimated standard errors using the jackknife method, and obtained the following results, as summarized in Table 2.

The penalty for failing to use VE in the presence of #version lines is the ratio of θ_{HAND} to θ_{VE}, which indicates an increase of about 36% in the effort required to complete an MR. (This coefficient was statistically significant at the 5% level). Restated, if a developer performs three changes to code involving #version lines in a given amount of time without VE, the same developer using VE could perform, on the average, four changes of the same size and type to the same code. At the same time, changes performed using VE were of the same difficulty (requiring a statistically insignificant 7% increase in effort) as changes with no #version lines at all (θ_{VE} versus θ_{NONE}).

We were successful in selecting similar developers: the ratio between the largest and smallest developer coefficients was 2.2, which would mean that the least efficient developer would require 120% additional effort to make a change compared to the most efficient developer, but

Factor	Estimates	Significance
Developer	$\frac{\max_i \alpha_i}{\min_i \alpha_i} = 2.2$	Not significant
Type	$\frac{\beta_{BUG}}{\beta_{NEW}} = 1.26$	Significant, $p = 0.06$
	$\frac{\beta_{CLEANUP}}{\beta_{NEW}} < 1$	Not significant
	$\frac{\beta_{INSP}}{\beta_{NEW}} < 1$	Not significant
Size	$\gamma = 0.19$	Not significant
VE Use	$\frac{\theta_{HAND}}{\theta_{VE}} = 1.36$	Significant, $p = 0.05$
	$\frac{\theta_{VE}}{\theta_{NONE}} = 1.07$	Not significant

Table 2: Results from model fitting.

Factor	Estimates	Significance
Developer	$\frac{\max_i \alpha_i}{\min_i \alpha_i} = 2.2$	Not significant
Size	$\gamma = 0.15$	Not significant
VE Use	$\frac{\theta_{HAND}}{\theta_{VE}} = 1.36$	Significant, $p = 0.1$
	$\frac{\theta_{VE}}{\theta_{NONE}} = 1.07$	Not significant

Table 3: Results for a model with no type factor and a random assignment of VE factor for MRs contataing delta done with and without the use of VE.

the jackknife standard errors indicated that a difference of this size was not large enough to be distinguishable from random fluctuations (i.e. there was no statistically significant evidence that the developers differed). This fact indicates that we were successful in selecting "similar" developers for our sample.

The type of a change was a significant predictor of the effort required to make it, as bug fixes were 26% more difficult than comparably sized additions of new functionality. Improving the structure of the code, the third primary reason for change (see, for example, [26]) was of comparable difficulty to adding new code, as was a fourth class of changes, implementing code inspection suggestions.

The variable γ in Equation (1) was estimated to be 0.19. That is, the size of a change did not have a particularly strong effect on the effort required to make it.

Validity of the results

To ensure that the estimated effects were valid, a number of steps were taken.

First, we took a conservative approach (under the null hypothesis) to mark all changes that contained deltas with the VE signature and without the VE signature as done by hand. To verify that this choice had no effect on the results, we repeated the analysis but randomly assigned each change one of the two conditions (see Table 3).

Second, we selected a balanced set of developers with similar change profiles to reduce inherent variability in developer performance. This was achieved by choosing developers who were actively changing the code in the considered six year period (1990 to 1995) and making similar numbers of changes (300 to 500) in that period.

Third, we made sure the tool effect would be identifiable from the sample given other key factors affecting the change effort - size, type, and developer. In linear regression, this is referred to as checking for collinearity. Ignoring such relationships could lead to situations where the tool effect would be indistinguishable from other factors affecting change effort.

We first checked for interactions between the developer and VE usage. Such interaction occurs frequently (developers tend either to use VE or not to use VE). From the set of developers selected in the second step we chose only those that had similar numbers of changes with and without VE and performed at least 40 changes under each condition. This brought us to the final sample of 9 developers we used in the analysis.

The relationship between the tool usage and the size of a change was insignificant. However, the interaction with the type of change was strong. New code was more likely to be done without VE, while bug fixes were more likely to be done with VE. This interaction confounds the tool effect with a factor known to influence the difficulty of a change. However, the interaction works in favor of the null hypothesis - bug fixes require more effort and are more often done using VE.

To verify that the interaction is not affecting the results we fitted the model with no factor for the type of change. In addition, the MRs that contained both VE and HAND deltas were randomly marked as being done by hand or with VE. The results are in Table 3. The estimated VE coefficient did not change from the original model in Table 2, but the variance of the estimate increased because of the additional variability caused by not adjusting for the change type factor.

Fourth, we validated the model using the jackknife method. We compared the effect of VE for changes that have similar values of the primary cost drivers (developer, size of change, type of change). These drivers were found to affect the effort significantly in [10]. Using the jackknife, we measured the significance of the effects given by the model. More details on validation, the model fitting and the algorithm are in [10].

Despite all these checks, the results warrant some caution. Although the selected developers performed similar numbers of changes with and without the tool, the pattern was not independent of time. Eight out of nine developers gradually moved towards exclusive usage of the tool, while one abandoned usage of the tool over the considered period. Because of this imbalance, the tool usage factor is confounded with time and other factors such as natural decay of the software architecture. Because of the nature of the observational study, other confounding factors might be present despite all the precautions.

6 A General Framework for Evaluating Software Tools

In this section, we consider how to generalize the process used in our case study to other software development environments and software tools.

In our case study, the effort analysis (Sections 4 and 5) made use of generic change data that are present in any modern version control system (as described in Section 2). Thus, the repeatability of our experiment in other settings relies primarily on the ability to correlate tool usage with change history. The particulars of the VE tool provided a very direct link between tool usage and changes, for two reasons:

- VE is an editor and so is used directly to make changes to the software;
- VE leaves a trace because of the trailing white space it inserts at the end of #version lines.

In general, we cannot expect to be so fortunate. Many software tools, such as debuggers, source code analyzers, profilers, etc., are not editors. They are used to examine and analyze software source but not to modify it. This is not terribly problematic, since software tools can be instrumented to record when they are applied to a software entity. Of greater importance, the VE trace is a direct causal link between the use of VE and the change history; we cannot expect to find such a direct link for all software tools. Instead, we must rely on temporal locality as a substitute for causality. That is, we must assume that a change made to software entity e at time t by developer d is (partially) aided by software tools that developer d applied to e (or entities related to e) in some window of time before t. This assumption is quite reasonable for many software tools such as error detectors and debuggers, though it may not apply as well to general program comprehension tools which could be used far before a change is made.

This leads us to the following general process:

1. Via automated non-intrusive monitoring, record the tools a developer uses in the course of software development and the software entities to which they were applied.
2. Correlate the monitoring information recorded in step 1 to the modifications to software entities that are recorded by the version control system, using temporal locality to link the application of a software tool to entity e (and related entities) to modifications to e.
3. Use the effort analysis algorithm of Section 4 on the data from steps 1 and 2 to estimate how the use of the tool affected developer effort, code quality, interval, etc.

As described in Section 5, it is important to control confounding variables such as developer experience and type of change in the above process. In other environments, additional variables may come into play.

This approach could be used to evaluate new tools as well as existing tools. To assess the impact of a new tool (or an enhancement of an existing tool) the usage data have to be collected from the set of developers who use the tool before the large scale deployment. When the effects of the tool usage become apparent the tool may be recommended for the wide-scale deployment. The effects should be estimated by comparing the changes done by the developers before and after the introduction of a tool.

The approach should work well for organizations where developers work on a single project at a time until completion. In some organizations the code changes are recorded in the version control system only at the time of completion. In such cases the start of an MR should be recorded as the date of completion of the previous MR done by the same developer. In organizations where developers work on multiple projects simultaneously the approach might require more substantial modifications.

7 Related Work

There is a substantial amount of work on evaluating software tools, which falls into three broad categories: controlled experiments on software tool use, software tool assessment, and case studies of software tool use. We also review related work on effort estimation in software projects.

Controlled Experiments on Software Tool Use

Controlled experiments on software tools typically use two groups to evaluate a tool on a given task: a study group that uses the tool and a control group that does not use the tool. Such experiments have been done on program slicing tools [14], algorithm animation tools [13], and structured editors [19], to name but a few. The study of Ormerod [19] is interesting because of the detailed level of tool instrumentation: a log of all

keystrokes entered into the structured editor for Prolog was recorded and used to identify edits, edit times, and errors made. There is a huge body of work in the Human Computer Interaction community that deals with the related issue of user interface design and evaluation. Many such studies evaluate analyze how different user interfaces affect task performance [9, 24].

Of course, our study is not a controlled experiment, although we did attempt to control for developer variability (see Section 5). Our work is a mix of a case study and a quasi-experiment. We have analyzed historical project data (tool usage analysis, time sheet data, and version control data), controlling for confounding variables, and have defined a general framework so that others can use our approach in different settings.

Software Tool Assessment

Software tool assessment is an industry of substantial size. As summarized by Poston and Sexton [21], the software tool assessment process consists of the following basic steps:

1. identifying and quantifying user needs;
2. establishing tool-selection criteria;
3. finding available tools;
4. selecting tools and estimating the return on investment;
5. acquiring a tool and customizing it to better fit the environment;
6. monitoring of tool usage to determine the impact of a tool.

Many tool assessment processes and standards (such as IEEE Standard 1175) focus on the use of forms to gather data to guide the first five steps of the above process [18, 21]. These include forms for needs analysis, tool-selection criteria, tool classification, and tool-to-organization and tool-to-tool relationships. Our work complements such work by addressing the final point (6) above. We use a highly-automated technique combining tool usage information with change effort analysis to estimate the impact of a tool in an organization.

Brown and Wallnau [5] present a framework for evaluating software technology. They observe that:

> Technology evaluations are generally ad-hoc, heavily reliant on the evaluation staff's skills and intuition.

Their framework is based on the idea of "technology deltas", by which they mean two things: how one tool differs from another, and how the differences between tools address specific needs. In our case study, the "delta" between VE and a standard text editor is the ability to manage certain pre-processor directives for the developer.

Case Studies

Kitchenham, Pickard and Pfleeger present a framework and guidelines for performing case studies of software tools and methods [12]. They observe that:

> A case study is usually preferable to a formal experiment if the effect of the change [the introduction of a new tool or change to a process] cannot be identified immediately. For example, if you want to know if a new design tool increases reliability, you may have to wait until delivery to assess the effect on failures.

It is exactly this scenario that our work addresses, as it makes use of historical data to identify the impact of a tool over some period of time. Exactly how long one needs to collect data in order to make such an assessment is an open question.

Bruckhaus et al [6] present a case study of how requirements-management tools affected the productivity of requirements planners, across several projects. Their goal was to find which projects would benefit from new tools. In this study, they measured productivity (after the fact) by the ratio of the number of features in a project to total effort expended in the project (number of minutes). They examined how the presence/absence of a tool, project size and software process (simple or complex) affect productivity. Measuring at this macro level makes it difficult to separate the impact of the tool from other confounding variables (such as experience, and size of the feature). Project and process could be included as factors in our model.

Effort Estimation

Previous work on developing models of effort (of which a recent example is [23]; see also its references) has dwelt on predicting the effort that will be required to complete a nascent project. The COCOMO model [4] and function points [1] are frequent contributors to these predictions. Our problem is substantially different as it works with smaller changes (MRs as opposed to projects). Also, we derive estimates of the effort that was required for changes that were part of already completed projects instead of concentrating on prediction.

Acknowledgments

This research was supported in part by grants SBR-9529926 and DMS-9208758 to the National Institute of Statistical Sciences.

REFERENCES

[1] A. J. Albrecht and J. R. Gaffney. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Trans. on Software Engineering*, 9(6):638–648, 1983.

[2] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In *Proceedings of the 8th Conference on Software Configuration Management (SCM-8)*, pages 146–157. Springer-Verlag, LNCS 1439, 1998.

[3] V. Basili and R. Reiter. An investigation of human factors in software development. *IEEE Computer*, 12(12):21–38, December 1979.

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

[5] A. Brown and K. Wallnau. A framework for evaluating software technology. *IEEE Software*, pages 39–49, September 1996.

[6] T. Bruckhaus, N. Madhavji, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Software*, pages 29–38, September 1996.

[7] J. O. Coplien, D. L. DeBruler, and M. B. Thompson. The delta system: A nontraditional approach to software version management. In *International Switching Symposium*, March 1987.

[8] B. Curtis. Substantiating programmer variability. In *Proceedings of the IEEE 69*, July 1981.

[9] H. Gottfried and M. Burnett. Programming complex objects in spreadsheets: An empirical study comparing textual formula entry with direct manipulation and gestures. In *Proceedings of the Seventh Workshop on Empirical Studies of Programming.* Ablex Publishing Co., 1997.

[10] T. L. Graves and A. Mockus. Inferring change effort from configuration management data. In *Metrics 98: Fifth International Symposium on Software Metrics*, pages 267–273, Bethesda, Maryland, November 1998.

[11] T. L. Graves and A. Mockus. Identifying productivity drivers by modeling work units using partial data. *Technometrics*, 1999. submitted.

[12] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, pages 52–62, July 1995.

[13] A. Lawrence, A. Badre, and J. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 48–54, October 1994.

[14] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the First Workshop on Empirical Studies of Programming,* (June 1986). Ablex Publishing Co., 1986.

[15] P. McCullagh and J. A. Nelder. *Generalized Linear Models, 2nd ed.* Chapman and Hall, New York, 1989.

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

[17] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. Submitted to ACM Transactions on Software Engineering and Methodology.

[18] V. Mosley. How to assess tools efficiently and quantitatively. *IEEE Software*, pages 29–32, May 1992.

[19] T. Omerod and L. Ball. An empirical evaluation of TEd, a techniques editor for prolog programming. In *Proceedings of the Sixth Workshop on Empirical Studies of Programming.* Ablex Publishing Co., 1996.

[20] A. Pal and M. Thompson. An advanced interface to a switching software version management system. In *Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, July 1989.

[21] R. Poston and M. Sexton. Evaluating and selecting testing tools. *IEEE Software*, pages 33–42, May 1992.

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

[23] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Trans. on Software Engineering*, 23(12):736–743, November 1997.

[24] B. Shneiderman. *Designing the User Interface (2nd Edition).* Addison-Wesley, 1991.

[25] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[26] E. B. Swanson. The dimensions of maintenance. In *2nd Conf. on Software Engineering*, pages 492–497, San Francisco, California, 1976.