Measuring Technology Effects on Software Change Cost

D. L. Atkins, A. Mockus, and H. P. Siy

May 11, 2000

Abstract

We describe a methodology for precise quantitative measurement of technology impact on software change effort. The methodology employs measures of small software changes to determine the effect of technology. We illustrate this approach in a detailed case study on the impact of using two particular technologies–a version-sensitive source code editor and a domain engineered application environment–in a telecommunications product. In both cases the change effort was reduced. The methodology can precisely measure cost savings in change effort and is simple and inexpensive since it relies on information automatically collected by version control systems.

1 Introduction

Software engineering productivity is notorious for being difficult to improve [6]. New technologies are constantly being introduced in the hopes of increasing productivity by making software easier to develop. While they have the potential greatly to improve the quality and maintainability of software, deploying and maintaining a new technology in a large organization can be an expensive proposition. We explore how to quantify the effects of existing software technologies in ongoing large-scale software projects, presenting a simple methodology that correlates technology usage with effort estimates based on analysis of the change history of a software project.

Quantifying the impact of a technology on software development is particularly important in making a case for transferring new technology to the mainstream development process. Rogers [17] cites observability of impact as a key factor in successful technology transfer. Observability usually implies that the impact of the new technology can be measured in some way. Most of the time, the usefulness of a new technology is demonstrated through best subjective judgment. This may not be persuasive enough to convince managers and developers to try the new technology.

The main assumption of our work is that a major effect of a software technology is to make it easier or simpler for a developer to make certain modifications to a software entity. The focus of our methodology is the analysis of changes to the software. First, we obtain a number of change measures, such as size, purpose, developer identity, and technology usage, from the change history of the source code. Then we quantify the impact of these measures on effort it takes to complete a change using the algorithm introduced in [11]. Finally, we adjust for the possible differences between changes done with and without the technology. We make this adjustment based on the value added by an average change. The value may be expressed in terms of new features introduced by the set of changes or it might reflect some other enhancement of software.

As we will see, this process is largely automatic, inexpensive, non-intrusive, and applicable to most software projects using version control systems. Furthermore, it can be applied to an entire software project in its actual setting as we do here to measure the effects of a version-sensitive source code editor and of a domain engineered application environment on software change effort. Despite fairly simple general features, there are a number of differences between the ways the methodology is applied to estimate the effect of various technologies. The goal of this paper is to highlight and summarize these differences to make the methodology easier to use in practice.

We start by briefly describing the software project under study, software changes, and data sources in Section 2. Section 3 describes the two technologies under consideration. Section 4 describe step-by-step application of our methodology. Finally we conclude with a relevant work section and a summary.

2 Background

The case study here revolves around a large telephone switching software system developed over two decades. Lucent Technologies' 5ESS™switch is used to connect local and long distance calls involving voice, data and video communications. The 5ESS source code is organized into subsystems with each subsystem further subdivided into a set of modules. Each module contains a number of source code files. The change history of the files is maintained using the Extended Change Management System (ECMS) [12], for initiating and tracking changes, and the Source Code Control System (SCCS) [16], for managing different versions of the files.

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 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 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 language abstract associated with it, provided by the developer, 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 [14]. In the analysis below, we distinguish defect fixes from other types of maintenance activities based on the presence of appropriate words in the one line English text MR abstract recorded by ECMS.

We also obtained a complete list of identifiers of MRs that were done using domain engineered application environment and/or using version sensitive editor. Thus, for each MR, we were able to obtain the following information,

- who made the change (developer login)
- size of the change (number of lines added and deleted)
- number of deltas
- duration (dates of first and last deltas)
- purpose of change
- number of files touched
- whether it was done using the technology under consideration.

3 Applications

In this section we describe two technologies we evaluate. The first one represents a source code editor that is designed to show a desired version of the source code. The second example describes a domain engineered application environment including a special language and a GUI based programming interface.

3.1 VE: A Version-sensitive Editor

The Version Editor (VE) is used by 5ESS developers to simplify the view of source code as they make changes. The software project for these programmers requires the concurrent development and maintenance of many sequential versions as well as two main variants for domestic and international configurations of the product. The 5ESS source code may be common to more than two dozen distinct releases of the code, which may be deployed products in maintenance mode, or new product versions under active development.

As described in [2], the software releases form a complex version hierarchy with the often conflicting project management goals of isolating deployed releases from current development changes yet maximizing commonality to promote the automatic flow of software fixes to future releases. Since the industrial source code management technology of the early 1980's did not have good support for branching and merging, source code was kept common among many releases with release specific differences delineated by a special embedded #version directive. This directive is similar to

```
  ...
  routing = GetRoute())
#version (4A)
  dest = GetDest(routing);
  if (dest.port == 0)
    return(ConnectLocal(routing));
#endversion (4A)
  Connect(routing);
  ...
```

```
  ...
  routing = GetRoute())
#version (4A)
  dest = GetDest(routing);
#version (! 5A)
  if (dest.port == 0)
#endversion (!5A)
#version (5A)
  if (dest.port == 0 || dest.module == 0)
#endversion (5A)
    return(ConnectLocal(routing));
#endversion (4A)
  Connect(routing);
  ...
```

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

a C preprocessor #if where a symbol (corresponding to the release) is used for control and the symbol may be negated.

This system permits a single source file to be extracted to produce a different version for each software release. Software development environment tools verify the consistent use of these constructs according to a release hierarchy maintained by the system and perform the extraction of the source code for building each software release. For example, the first frame in Figure 1 shows a source file where three lines of code are specific to the 4A release. The system guarantees that these lines will not appear in earlier releases but will appear in later releases. Also, the lines will not appear in isolated releases (the domestic and international configurations are all isolated from each other).

A developer adding new code must target the change for a specific release and then bracket it by the appropriate #version constructs. When existing code is changed, it must be logically deleted with a #version construct using the negation of 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.

This simple example shows how even a one line code change requires the developer to add five lines to the file (four control lines and the changed code line). In addition to this extra overhead for a logical one line code change, the version control lines make the source file more difficult to read and understand. In the project being studied there are several dozen distinct releases and 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.

The version-sensitive editor VE [8, 15, 3] was made available to make this situation more manageable for the developer. This tool allows the developer to edit in a view that shows only the code that will be extracted for the release being changed and performs the automatic insertion of any necessary control lines.

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. The developer only has to use standard **vi** or **emacs** editing commands, and VE inserts the required #version directives (behind the scenes).

The use of VE by developers is entirely optional. The usage of VE may be detected, because VE leaves a signature on all of the #version/#endversion control lines that it generates. (See [2] for more details.) Thus we can 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 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

```
  routing = GetRoute(routing))
  dest = GetDest(routing);
▯ if (dest.port == 0 || dest.module == 0)
    return(ConnectLocal(routing));
  Connect(routing);

MR 12467 by dla,97/9/21,assigned [Local routing]
Versioning: 5A inside 4A
”route.c” [modified] line 67 of 241
```

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

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.

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.

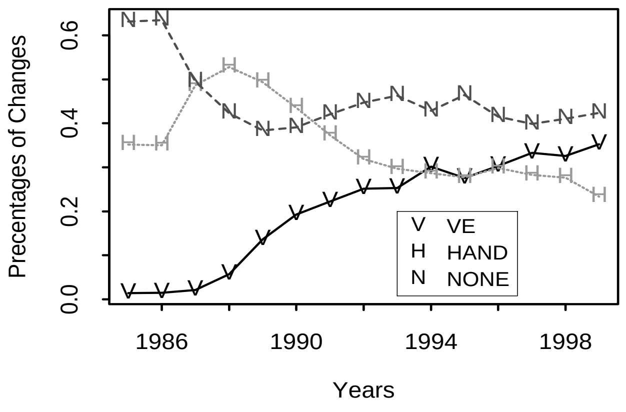

Figure 3: VE usage over time.

3.2 Domain Engineering

Traditional software engineering deals with the design and development of individual software products. In practice, an organization often develops a set of similar products, called a family or product line. Traditional methods of design and development don't provide formalisms or methods for taking advantage of these similarities. As a result the developers practice some informal means of reusing designs, code and other artifacts, massaging the reused artifact to fit into new requirements. This can lead to software that is fragile and hard to maintain because the reused components were not meant for reuse.

Domain Engineering (DE) [19, 7, 9] approaches this problem by defining and facilitating the development of software product lines rather than individual software products. This is accomplished by considering all of the products together as one set, analyzing their characteristics, and building an application engineering environment to support their production. In doing so, development of individual products (henceforth called Application Engineering) can be done rapidly at the cost of some significant up-front investment in analyzing the domain and creating the environment.

The process is summarized in Figure 4. In this figure, DE is further divided into domain analysis and domain implementation and integration. Domain analysis identifies the commonalities among members of the product line as well as the possible ways in which they may vary. Usually, several domain experts assist in this activity. Also, the application engineering environment is designed and built. This usually involves creation of a domain-specific language as well as a graphical user interface front end as well as a source code generator back end. Domain implementation and integration deploys the DE-based process, making necessary adjustments to product construction tools (makefiles, version control systems, etc.) and to the overall development process.

DE proponents believe that practitioners can improve the productivity of application engineering by a factor of between two and ten, although there has been no quantitative empirical support for such claims.

Several teams have used the DE-based process to reengineer specific domain areas within the 5ESS software [1]. We conducted a study to evaluate the impact of the AIM project, a DE effort to reengineer the software and the process for developing the multiplicity of screen interfaces to the 5ESS switch database.

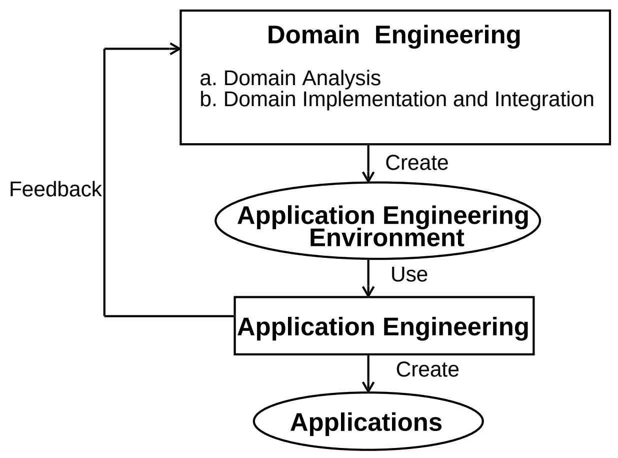

Figure 4: The Domain Engineering-based development process is an iterative process of conducting Domain Engineering and Application Engineering.

The problem faced by the screen developers was that most clients who purchased 5ESS required customization of their screen interfaces. In the old process, screens were customized by inserting `#ifdef`-like compiler directives into existing screen specification files. Over time, the specification files have become difficult to maintain and modify.

The AIM project used DE to identify commonalities and variabilities in different clients' interface requirements. These results provided input to the development of a GUI tool for assisting in the design of and keeping track of the customized screens. Information gathered through the GUI was saved in files whose format was specified by a domain-specific language. During the product build process, a code generator would then take these files and generate the screen specification files.

More details on the AIM study is published in an earlier paper [18]. In some sense, the problem here is not unlike the problem addressed by VE which facilitates the maintainance of multiple versions of code. However, the creators of AIM undertook a higher level, domain-specific solution in an attempt to achieve even higher productivity.

4 Methodology

We outline here a general framework for analyzing the impact of a software technology on software development effort. The main assumption of the framework is that if the technology affects the development effort, the developers using it would be able to perform more difficult changes per unit time.

The estimation is done at an individual change (MR) level using algorithm introduced in [11]. The algorithm uses change measures, such as the person doing the change and change size and purpose, to quantify their contribution to the change effort. To assess the impact of the technology we include technology usage among other predictors of change effort and use the algorithm to quantify its impact. For certain technologies, such as VE, this step is sufficient, because the technology does not modify the definition of the change itself. The VE does not appear to modify the definition of the change because the same development process is used and the same parts of the system are changed with and without the tool.

The AIM case is different in that the changes are done not by editing the source code, but by drawing graphs in a GUI environment. This suggests that the change size may not affect effort in the same way for AIM and for non-AIM changes. It can be easily addressed by omitting predictors that are not comparable across technology usage. However, some technologies might affect the definition of the change in some more fundamental ways. To deal with this issue we need to quantify the ratio of value added by an average change that uses and that does not use the technology and adjust the effort savings correspondingly. There are several ways to assign value to software changes, the most obvious being the amount of or the revenue from the new functionality introduced by the change.

The analysis framework consists of five main steps.

1. Obtain measures of changes. Identify the changes made to the software entity of interest and whether or not the technology was used.

2. Select a subset of these measures to predict the effort to make a change. The minimal subset typically includes the identity of a developer, whether or not the technology was used, and the size and purpose of the change.

3. Check for collinearity among predictors. In particular, select subset of developers who have implemented a substantial number of changes with and without the technology. Since the developer factor often has the largest effect on change effort, a balanced group of developers reduces variances of other estimates. More importantly, it makes results insensitive to potential corre-

lation between use of technology and overall productivity of individual programmer.

4. Fit and validate a set of candidate models. The models that explain more variation in the data and have fewer parameters are preferred. The fitted models could be used to test the significance of the effect of technology and/or to predict effort savings.

5. Obtain and compare the average value added by a change that utilize the technology and a change that does not. Use this factor to adjust for the fact that the technology might affect the value added by an average change.

The following sections explain each step in detail.

4.1 Change Measures and Technology Use

The basic characteristic measures of software changes include: identity of the person performing the change; the files, modules, and actual lines of code involved in the change; when the change was made; the size of the change measured both by the number of lines added or changed, the number of deltas, and the number of files touched; and the purpose of the change including whether the purpose of the change was to fix a defect or to add new functionality. Many change management systems record data from which such measures can be collected using software described in [13].

The information on files modules and lines changed is usually sufficient to determine if the software entity of interest was touched by the change. The determination of technology involvement in the change might be more complicated. We first discuss how to determine if the technology was used and then if it was not used.

In real life situations developers work on several projects over the course of a year and it is important to identify which changes they performed using the technology of interest. There may be several ways to identify these changes. In our VE example the tool left a trace in the SCCS files. In the AIM example the domain engineered features were implemented in a specific set of code modules (we refer to them as AIM paths). In other cases we considered the technologies involved use of a new programming language. To identify if technology was used it was sufficient to determine the language used in the changed file. In yet another case, a new library was created to facilitate code reuse. To identify relevant changes in that case, we look at the function calls used in the modified code to determine if these calls involve the API of the new library.

Finally, to perform the comparison, we need to identify changes to a software entity that were done without to the use of technology. In case of VE the information was available directly from SCCS except for a subset of changes that had no #version lines. Consequently we had three types of MRs: changes done using VE, changes done without VE, and changes without #version lines. In the AIM example, the source code to the previously used screen specification files had a specific set of directory paths. We refer to those paths as pre-AIM paths. Based on AIM and pre-AIM paths sets of paths we classified all AIM MRs into three classes: MRs that touched at least one file in the AIM path, MRs that do not touch files in the AIM path, but touch at least one file in the pre-AIM path, and MRs that do not touch files in the AIM or pre-AIM paths. In both cases there are three categories of changes that we label:

TECH — MRs done on the software entity that involve use of technology;

no-TECH — MRs done on the software entity not involving the use of technology;

other — MRs done on other software entities or MRs where the use of technology is uncertain.

4.2 Variable Selection

First among variables that we recommend always including in the model is a developer effect. Other studies have found substantial variation in developer productivity [10]. Even when we have not found significant differences and even though we do not believe that estimated developer coefficients constitute a reliable method of rating developers, we have left developer coefficients in the model. The interpretation of estimated developer effects is problematic. Not only could differences appear because of differing developer abilities, but the seemingly less productive developer could have more extensive duties outside of writing code.

It was found [11, 18, 2] that the purpose of the change (as estimated using the techniques of [14]) also has a strong effect on the effort required to make a change. In these studies, changes that fix defects are more difficult than other comparably sized changes.

Naturally, the size of a change may have a strong effect on the effort required to implement it. We have chosen the number of lines added, the number of files touched, and the number of deltas that were part of the MR as measures of the size of an MR. It is important to note that in the AIM case, the changes involving technology were done using a special GUI environment instead of editing the source code in individual files. This suggests that the size of AIM changes might have a different effect on effort than the size of other changes. One suggestion of that is the fact that the AIM changes tend to be larger [18]. This potential problem lead us to excluded the size predictors from some of the AIM effort models.

Finally, we include the variables of interest that concern the use of technology. As we described above, that indicator has three levels: technology was used, technology was not used, and the use was uncertain or the change was done on a different software entity.

4.3 Collinearity and Developer Selection

Statistical regression models typically require that the predictors used should not be collinear, that is, they should be indpendent of each other. All change size measures tend to be strongly correlated. Here we chose to use only the number of deltas as our change size predictor. The number of deltas is correlated both with the number of files touched (for each file touched there must be at least one delta) and with the number of lines added (developers are typically unable to add large pieces of code within a day and they usually check in the code each day, resulting in multiple deltas for changes with many lines). Consequently, the number of deltas is a good overall predictor of change size.

Since the developer identity is the largest source of variability in software development (see, for example, [4, 10]), we first select a subset of developers that had a substantial number of MRs both with and without using technology so that the results would not be biased by the developer effects. If we had some developers that made changes exclusively with the aid of technology, the model could not distinguish between their productivity and the technology effect.

To select developers of similar experience and ability, we chose developers that had completed between 150 and 1000 MRs on the considered product in their careers. We chose only developers who completed at least 40 VE MRs for the VE analysis and at least 15 AIM MRs for the domain engineering analysis. The resulting subset contained nine developers in the VE case and ten developers in the AIM case.

4.4 Models and Interpretation

In the fourth step we are ready to fit the models and interpret the results. The models are fit and the significance values for the predictors are calculated as described in [11]. The decisions made in previous steps suggested two models. The first model assumes that the change size properties (number of deltas) have the same effect on developer effort whether or not the technology is used. There is little basis to doubt the assumption in the case of VE, since the same source code files are changed with and without the VE and the additional editing features provided by the VE do not support or inhibit proliferation of deltas. In the AIM case, the changes are made through GUI interactions rather than by directly editing the source code files. Also, the underlying language changed. Consequently, it is important to exclude the assumption that the change size influences the effort in the same way. We address that by fitting the second model without any change size predictors.

The models are:

$$\begin{aligned} E(\text{effort}) &= \#\text{delta}^{\alpha} \times \beta_{\text{Fix}} \times \gamma_{\text{TECH}} \times \gamma_{\text{no-TECH}} \\ &\quad \times \prod_i \delta_{\text{Developer}_i} \\ E(\text{effort}) &= \beta_{\text{Fix}} \times \gamma_{\text{TECH}} \times \gamma_{\text{no-TECH}} \times \prod_i \delta_{\text{Developer}_i}, \end{aligned}$$

where in the model formula we use β_{Fix} as a shorthand for $\exp\left(I(\text{is a defect fix}) \log \beta_{\text{Fix}}\right)$, where $I(\text{is a defect fix})$ is 1 if the MR is a defect fix and 0 otherwise. The same abbreviation is used for γ and for δ. The basis for comparison was set by letting $\beta_{\text{NotFix}} = \gamma_{\text{other}} = 1$. The estimated coefficients with p-values and 95% confidence intervals are calculated using a jackknife method (for details see [11]) and are given in Table 1. The coefficients reflect how a particular factor multiplies the change effort in comparison with the base factors β_{NotFix} and γ_{other} which are assumed to be 1 for reference purposes.

The estimates of technology effects in Table 1 are very similar across models. However, the first model is a more precise description of the VE case, while the second model is preferable for the AIM case (because the change size is likely to have a different influence on effort in the AIM environment, while this is not true for the VE case).

A large difference exists between the relative impact of each technology. Not using VE increases average MR effort about 40% $\left(\frac{\gamma_{\text{no-TECH}} - \gamma_{\text{TECH}}}{\gamma_{\text{TECH}}} \approx 0.4\right)$ and using AIM decreases that effort three to four times ($\frac{\gamma_{\text{no-TECH}}}{\gamma_{\text{TECH}}} \approx 3.5$).

There are important differences between the *other* MRs in the two models. In the AIM case, these MRs are done on a different software entity, but are not different from the *no-TECH* MRs in any other way. In the VE case, these MRs did not contain #version lines and so it was impossible to determine if the VE was used. Furthermore, such MRs are likely to touch newer code (without the proliferation of #version lines) and be much smaller. Consequently, γ_{TECH} and $\gamma_{\mathrm{no-TECH}}$ are both larger than unit in the VE case.

Finally, the estimates indicate that the defect repair changes require 50 to 100 percent more effort than comparable changes of other types ($\beta_{\mathrm{Fix}} \approx 1.5$ for the VE subset of changes and $\beta_{\mathrm{Fix}} \approx 2$ for the AIM set of changes).

In the analysis above we have accounted for the possibility that the change size might differently affect the MR effort when technology is used. A more serious issue is that the technology might affect the functionality delivered by the average change. The next section discusses how to account for this extra effect of the technology.

4.5 Calculating Value Added by an Average Change

The models above provide us with the amount of effort spent at the fine MR level. More radical technologies like application engineering environment, might change the definition of MR so that MRs no longer implement the same functionality (or some other value added to the software entity) as before. We do not have any reason to believe that a tool like VE could have such an effect. However, in the AIM case we chose to adjust the effort savings for an average MR accordingly. More formally, let f_{TECH} be the functionality implemented by an average MR using the technology, while $f_{\mathrm{no-TECH}}$ the functionality implemented by an average MR that does not use the technology. Then the ratio of efforts to implement the same amount of functionality without and with the technology would be:

$$\frac{\gamma_{\mathrm{no-TECH}}}{\gamma_{\mathrm{TECH}}}\frac{f_{\mathrm{TECH}}}{f_{\mathrm{no-TECH}}} \qquad (1)$$

In the particular product the new functionality was called software features. Features add value to the software because they generate revenue and enhance competitiveness of the product. We assume that on average, all features implement similar amount of value. This is a reasonable assumption since we have a large number of features under both conditions and we do not have any reason to believe that the definition of a feature changed over the considered period. Consequently, even a substantial variation of functionality among features should not bias the results.

We determined the software features for each MR (MR implementing new functionality) using an in-house database. We had 1677 AIM MRs involved in implementation of 156 distinct software features and 21127 pre-AIM MRs involved in implementation of 1195 software features, giving 11 and 17 as the MR per feature ratio. Based on this analysis an average AIM MR implement about 60 percent more functionality than the average pre-AIM MR.

To estimate the total effect of the technology we need do integrate these effort savings over all changes and convert effort savings to cost savings. Also, we need to subtract the cost of creating and maintaining the technology itself.

The functionality in 1677 AIM MRs would approximately equal the functionality implemented by 2650 pre-AIM MRs. The effort spent on 1677 AIM MRs would approximately equal the effort spent on 420 hypothetical pre-AIM MRs using the estimated 75% savings in change cost obtained from the models above. This leaves total cost savings expressed as 2230 pre-AIM MRs.

To convert the effort savings from pre-AIM MRs to technical head count years (THCY) we obtained the average productivity of all developers in terms of MRs per THCY. To obtain this measure we took a sample of relevant developers, i.e., those who performed AIM MRs. Then we obtained the total number of MRs each developer started in the period between January 1993 and June 1998. No AIM MRs were started in this period. To obtain the number of MRs per THCY, the total number of MRs obtained for each developer was divided by the interval (expressed in years) that developers worked on the product. This interval was approximated by the interval between the first and the last delta each developer did in the period between January 1993 and June 1998. The average of the resulting ratios was 36.5 pre-AIM MRs per THCY.

Using MR per THCY ratio the effort savings expressed as 2230 pre-AIM MRs are equal to approximately 61 technical head count years. Hence the total savings in change effort would be between $6,000,000 and $9,000,000 in 1999 US dollars. This assumes technical head count costs varying between $100K and $150K per year in the Information Technology industry.

The total expense related to the AIM DE effort has been estimated to be 21 THCY. Based on our calculated effort savings, the first nine months of applying AIM saved around three times (61/21) more effort than was spent on implementing AIM itself.

We also compared our results with effort savings predictions documented in the business cases for AIM and several other DE projects performed on the 5ESS software. Our results were in line with the approximate predictions given in these documents indicating that DE interval reduction to one third to one fourth of pre-DE levels.

5 Related Work

The framework to evaluate effects of a tool is described in [2]. The methodology to assess the impact of Domain Engineering application environments is given in [18]. In this paper we extend and unify both frameworks to create a general approach for evaluating the impact of any software technology. We focus on practical applications of the approach by performing a detailed step-by-step analysis of two types of new technology.

This technique is very different in approach and purpose from traditional cost estimation techniques (such as COCOMO and Delphi [5]), which make use of algorithmic or experiential models to estimate project effort for purposes of estimating budget and staffing requirements. Our approach is to estimate effort after actual development work has been done, using data primarily from change management systems. We are able to estimate actual effort spent on a project, at least for those phases of development that leave records on the change management system. This is useful for calibrating traditional cost models for future project estimation. In addition, our approach is well-suited for quantifying the impact of introducing new technology to existing development processes.

6 Summary

We present a methodology to obtain cost savings from use of a software technology exemplified by a case study of a tool and an application engineering environment. We find that the change effort is reduced about 40% in the VE tool example and about four times in the application engineering environment example. Although both technologies were a success, the differences in relative savings are not surprising. The more radical change involving a GUI based programming environment can have correspondingly more significant impact than the use of a tool assisting only in a more narrowly defined task. It suggests that the usage of even very effective tools may have impact of improving productivity on the order of 10%, while to achieve more significant savings a special programming environment to accomplish the frequently repeating set of tasks has to be created. Of course, creation of such environment may be more costly than the purchase and maintenance of a tool, so each technology has to be applied only with a careful consideration of its benefits.

The described methodology is based on automatically extractable measures of software changes and should be easily applicable to other software projects that use source code version control systems. Since most of the change measures are kept in any version control system, there is no need to collect additional data.

We described in detail all steps of the methodology to encourage replication. We expect that this methodology will lead to more widespread quantitative assessment of software productivity improvement techniques. We believe that most software practitioners will save substantial effort from trials and usage of ineffective technology, once they have the ability to screen new technologies based on a quantitative evaluation of their use on other projects. Tool developers and other proponents of new (and existing) technology should be responsible for performing such quantitative evaluation. It will ultimately benefit software practitioners who will be able to evaluate appropriate productivity improvement techniques based on quantitative information.

References

[1] M. A. Ardis and J. A. Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal*, 3(3):10–20, September 1998.

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the effectiveness of software tools. In *1999 International Conference on Software Engineering*, Los Angeles, CA, May 1999. ACM Press.

[3] 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.

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

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

[6] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[7] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998.

[8] 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.

[9] D.A. Cuka and D.M. Weiss. Engineering domains: executable commands as an example. In *Proc. 5th Intl. Conf. on Software Reuse*, pages 26–34, Victoria, Canada, June 2-6 1998.

[10] B. Curtis. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7):846, July 1981.

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

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

[13] A. Mockus, S. G. Eick, T. L. Graves, and A. F. Karr. On measurement and analysis of software changes. Technical report, Bell Laboratories, 1999.

[14] Audris Mockus and Lawrence G. Votta. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, San Jose, California, October 2000.

[15] 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.

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

[17] E. M. Rogers. *Diffusion of Innovation*. Free Press, New York, 1995.

[18] H. Siy and A. Mockus. Measuring domain engineering effects on software coding cost. In *Metrics 99: Sixth International Symposium on Software Metrics*, pages 304–311, Boca Raton, Florida, November 1999.

[19] D. Weiss and R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

Table 1: Model Coefficients

Model	Coefficient	VE estimate	VE p-val	VE 95% CI	AIM estimate	AIM p-val	AIM 95% CI
I	α	0.05	0.6	[-0.18,0.29]	0.5	0.000	[0.3,0.7]
	β_{Fix}	1.61	0.002	[1.2,2.1]	2.1	0.002	[1.4,3.1]
	γ_{TECH}	1.34	0.4	[0.7,2.6]	0.27	0.000	[0.17,0.44]
	$\gamma_{\mathrm{no-TECH}}$	1.94	0.007	[1.2,3.0]	1.07	0.68	[0.7,1.5]
II	β_{Fix}	1.58	0.002	[1.2,2.0]	1.98	0.002	[1.3,2.9]
	γ_{TECH}	1.34	0.35	[0.7,2.6]	0.29	0.002	[0.14,0.58]
	$\gamma_{\mathrm{no-TECH}}$	1.96	0.004	[1.3,3.0]	1.00	0.9	[0.7,1.5]