Software Support Tools and Experimental Work

Audris Mockus

April 5, 2007

1 Introduction

Presently it is difficult to imagine a software project without version control and problem tracking systems. This may be partly attributed to the emergence of open source projects where participants never meet each other and increasingly popular commercial globally distributed projects where groups of developers are often separated by many time zones. Version control and problem tracking tools are a basic necessity in such communication-poor environments. However, their value is getting more and more recognized in small co-located projects because information stored in these tools represent most of the project's decision making history.

Therefore, the idea to use repositories of software support tools to explain and predict phenomena in software projects and to create tools that improve software productivity, quality, and lead times appears to be promising. This is particularly salient to many open source software projects where all project related discussion and decisions are externalized in the mailing lists and other tools. In many such projects it is considered inappropriate to discuss project matters in private discussions not recorded on the relevant mailing lists.

The study of open source projects has other significant motivations as well. Such projects present a conundrum from software engineering perspective as they apparently lack key aspects such as requirements and design that are thought to be essential for project's success. The motivation of volunteer participants can not be convincingly explained using current economic theories. Open source is considered to be the new technological commons that, instead of being destroyed by over-use, is, on the contrary, benefiting from it. Tragedy of the commons is a concept of individually optimal decisions (for example, having a lager herd) leading to suboptimal outcomes for everyone (common grazing lands destroyed).

A typical analysis of software repositories includes retrieval, summarization, and validation of data from software project's version control and problem tracking systems. Unfortunately, extracting, cleaning and validating, and drawing conclusions from such data poses formidable challenges because data sources are not designed as measurement tools, and the tools involved as well as practices of using the tools vary from project to project.

The topic has recently attracted substantial attention including a special issue of the Transactions on Software Engineering (Vol. 31, No. 6) and an annual workshop on "Mining Software Repositories".

Here we attempt to outline the overall methodology and list some of the opportunities and challenges of using project support systems in empirical work. We start from the overview of the tools used, continue with methodology and its benefits, and conclude with the list of remaining challenges.

2 Tools supporting software development

Although software tools are used to support virtually any software development project there are important differences in the tools and the ways they are used. The first broad distinction can be made between open source projects and commercial projects. Open source projects tend to use three core tools. Version control systems such as CVS [3] (and now, increasingly, Subversion [4]) are used to keep track of the changes to the code and to grant permission to make changes to project members. Every change is usually accompanied by an automatic email sent to the special change mailing list to notify other project participants. Problem tracking systems, such as Bugzilla or Scarab, provide a way to report and track the resolution of issues. Finally, mailing lists provide a forum to discuss issues other than changes or problems. Most projects have a developer mailing list to discuss features, design, and implementation, and a user mailing list to discuss installation and usage of the product.

Commercial projects tend to contain more numerous and varied tools to track various aspects of the development and deployment processes. Although some of these systems contain little information helpful in analyzing software production this may change as the objectives and scope of the analyses evolve in the future. Sales and marketing support systems may contain customer information and ratings, purchase patters, and customer needs in terms of features and quality. The accounting systems that track purchases of equipment and services contain information about installation dates for releases. Maintenance support systems should have an inventory of installed systems and their support levels. Field support systems include information about customer reported problems and their resolution. Development field support systems contain software related customer problem tracking and patch installation tracking. Development support systems are similar to open source projects and contain feature, development, and test tracking. Common version control tools in commercial environments include ClearCase [25] and Source Code Control System (SCCS) [26] and its descendants. Most projects employ a change request management system that keeps track of individual requests for changes, which we call Modification Requests (MRs). Problem tracking systems, unlike change management systems, tend not to have built-in relationship between MRs (representing problems) and changes to the code. Extended Change Management System (ECMS) [14] is an example of change management system that uses SCCS for version control.

Large software products employ a number of service support tools to help predicting and resolving customer problems. Such systems may be used to model software availability [17], though their description is beyond the scope of this presentation.

3 Basic methodology

The amount and complexity of available data necessitates the use of analysis tools except, possibly, in the smallest projects. Such analysis systems contain the following capabilities [8]:

- Retrieve the raw data from the underlying systems via access to the database used in the project support tools or by "scraping" relevant information from the web interfaces of these systems. For example, CVS changes can be obtained via *cvs log* command, and Bugzilla data is stored in MySQL relational database.
- Clean and process raw data to remove artifacts introduced by underlying systems. Verify completeness and validity of extracted attributes by cross-matching information obtained from separate systems. For example, match changes from CVS mail archives, from *cvs log* command or matching CVS changes to bug reports and identities of contributors.
- Construct meaningful measures that can be used to assess and model various aspects of software projects.
- Analyze data and present results.

These capabilities represent processing levels that help users cope with complexity and evolution of the underlying support systems or the evolution of the analysis goals by separating these concerns into separate levels that can (and should) be validated independently. Each level refines data from the previous stage producing successively better quality data, however it is essential that links to raw data are retained to allow automatic and manual validation.

In summary, the key desirable features of the analysis include:

1. Iterative refinement of data with each iteration obtaining quantities that may be more interpretable and more comparable across projects.
2. Each item produced at every stage retains reference to raw data to facilitate validation.
3. Each processing level has tools to accomplish the step and validation techniques to ensure relevant results. Because the projects and the processes may differ, it is essential to perform some validation on each new project.

The main stages and the tools needed to perform them are described in the subsections below. More detail on tools used for open source projects may be found in [15].

3.1 Development process

The changes to the source code tend to follow a well-defined process. Unfortunately, that process may greatly vary with project, therefore it has to be obtained from project development FAQ or from another document on development practices. The practices need to be validated by interviewing several developers and testers (or other participants administering or using project support tools) on a small subset of their recent MRs or changes.

In rough terms, the new *software releases* or *software updates* are product deliveries that contain new functionality (features) and fixes or improvements to the code. *Features* are the fundamental design unit by which the system is extended. Large changes that implement a feature or solve a problem may be decomposed into smaller self-contained pieces of work often called *modification requests* (MRs).

To perform the changes a developer or a tester (or, in open source projects even the end user) can "open" MRs. The MR is then assigned (or self-assigned) to a developer who makes the modifications to the code (if needed), checks whether the changes are satisfactory, and then submits the MR. In some instances of development processes the tester or code owner may then inspect and approve the MR. The MR ID is usually included in a the comments or as a separate field in the attributes of the change recorded by the version control system.

Version control systems (VCS) operate over a set of source code files. An editing change to an individual file is embodied in a *delta*: conceptually the file is "checked out," edited and then "checked in." 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 such as the time the change was made, the person making the change, and a short comment describing the change. 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.

3.2 Retrieval of raw data

Software changes are obtained from version control systems and contain developer login, timestamp of the commit, change comment, file, and version id.

Once the set of revisions (and their attributes) are extracted, the underlying code changes can be extracted by obtaining all versions of all files and the differences between their subsequent versions. This is the most involved operation but it allows fully to reconstruct the code evolution in each file and is necessary to obtain the exact content of each change. The content of a change can then be used to determine if the change involved comments or code and to identify what functions or statements were changed.

Problem (or bug) reports typically contain MR ID, severity, development stage at which the problem was detected, software release, description, status history (identity of individuals, dates, and status changes), and various attachments needed to explain the nature of the problem or the way it was resolved. Unlike CVS, most problem tracking systems store information in a relational database. If access to such database is difficult to obtain, it may be possible to retrieve web pages for each problem report and extract the relevant attributes from these web pages.

Mailing lists tend to be archived and are, therefore, easy to download and process. Tools that identify and count threads, extract relevant dates, and patches may be helpful.

Extracting raw data, although time consuming, contains few pitfalls. However, different projects may use slightly different format or slightly different attributes even if they use the same systems. The most common issue is likely to be that network congestion or version control locking issues may prevent obtaining the full set of items. A more robust option is to retrieve data in smaller chunks.

3.3 Augmenting raw data

System generated artifacts involve data points that do not represent activity of interest. An example of such artifact is an empty delta where the code is not modified. Such changes are common when creating branches in the version tree. Another common problem is copying of CVS repository files. In such case, duplicate delta from two or more files that had the same origin or are included in several modules are eliminated.

MRs are groups of delta done to perform a particular task. In cases where MR is indicated in the comment of a delta such grouping can be established. Many projects using CVS do not follow that practice. A common approach is to group delta that share login and comment and are submitted within a brief time period. The drawback of this approach is difficulty of determining the right interval to create breaks and the possibility that delta with different comments may belong to the same MR. A sample of groupings needs to be inspected to determine the most suitable time window.

In cases where MR IDs (or other information needed for the analysis) are embedded within delta comment their ID has to be extracted from the comment and associated with appropriate delta. This tends to be fairly straightforward using pattern matching techniques because MR IDs have a well defined format.

The step of augmenting raw data may involve cleaning attribute values that are entered manually, because manual input is always associated with errors. If, for example, a release number is typed (instead of selected from a list of choices), it may be necessary to inspect all unique values (in their frequency order) and process the output to change at least the most frequent misspellings to their intended values.

3.4 Producing change measures

Before various measures are produced, the semantics of the attributes may need clarification. For example, each MR may be associated with a release where the problem was found and with all releases where it was fixed. Many large projects track the problem not just for a release where it was found but also for the future (and in rare cases for the past) releases where the problem may manifest itself. Typically, the problem is first resolved for the release where it has been reported because subsequent releases are often still in development stages. This has important implications for measurement. If we are analyzing the number of problems found in a release, we have to count such MR only once for the release it was detected in. However, if we are looking at effort and schedule, we have to investigate all MRs resolved in a particular release because it requires effort to resolve the same MR for each release. Therefore, MRs resolved for several releases would affect effort and schedule for each release.

Several change measures are described in [21]. Change diffusion measures the files, modules, and subsystems affected by the change or the developers and organizations involved in making the change. Change size may be operationalized as the number of lines of code (LOC) added or deleted and LOC in the files touched by the change. A convenient proxy for both size and diffusion is the number of delta in a change. The duration of a change may be measured conservatively as the time elapsed between the first and the last delta or by comparing MR creation and submission or resolution dates. Often it is helpful to separate bug fix MRs from MRs used to track new features [20] and identify MRs associated with customer reported problems. Measures of experience (number of delta) or productivity (number of MRs resolved per year) can be associated with a developer or organization and measures of faultiness (fraction of MRs reported by customers or post unit-test) can be associated with files or modules.

The choice of measures may be dictated by the needs of a particular study, but basic summaries tend to be useful in most studies.

3.5 Models and Tools

There has been a substantial amount of work involved in modeling software changes. It was found that past changes are the best predictor of module faults [9] and that the diffusion and expertise of developers affects the likelihood that a patch will break [21].

The idea that files frequently touched together but not with other files define chunks that can be maintained independently was investigated in the context of globally distributed software development in [22] and it was found that MRs spanning such chunks take longer to complete [11]. It turns out that MRs involving geographically separated developers take more than twice as long to complete [12].

Models of expertise and relevance can provide most relevant experts [19], most relevant files [27, 13], or most relevant defects [5].

A general topic of evaluating the effect of software engineering tools and practices relies on the ability to estimate developer effort [10]. Work in [1, 2, 7] investigates the effort savings of version sensitive editor, visual programming environment, and refactoring of a legacy domain.

A number of hypotheses on how open source and commercial software engineering practices differ and their effect on productivity quality and lead time are investigated in [18, 6].

At a much higher level entire releases are modeled via changes to predict release schedule [23]. The work assumes that a random number of fix MRs are generated with a random delay from each new feature MR. An assumption that the work flow of MRs in the past releases is similar to the work flow of the current release is used to predict release readiness [16].

A probability that a customer will observe a failure related to a software problem is modeled in [24].

4 Advantages and pitfalls of using project support systems

Probably the most obvious advantage of using project support systems such as customer problem tracking system is that the data collection is non-intrusive because such systems are already deployed and used. However, that does not reduce the need for in-depth understanding of a project's development process and, in particular, of how the support systems are used.

A long history of past projects whose data has been captured in project support systems enables historic comparisons, calibration, and immediate diagnosis in emergency situations.

The information obtained from the support systems is often fine grained, at the trouble ticket/software alarm/customer installation level. However, links to aggregate attributes, such as features and releases, is often tenuous.

The information tends to be complete, as every action involving development or support is recorded. However the information about what the action pertains to may be nontrivial to infer and some of the data entries, especially those not essential for the domain of activity, tend to be inconsistently or rarely supplied.

The data are uniform over time as the project support systems are rarely changed because they tend to be business-critical and, therefore, difficult to change without major disruptions. That does not, however, imply that the process was constant over the entire period one may need to analyze.

Even fairly small projects contain large volumes of information in the project support systems making it possible to detect even small effects statistically. This, however, depends on the extractability of the relevant quantities.

The systems are used as a standard part of the project, so the software project is unaffected by experimenter intrusion. We should note that this is no longer true when such data are used widely in organizational measurement. Organizational measurement initiatives may impose data collection requirements that the development organizations might not otherwise use and modify their behavior in order to manage the measures tracked by these initiatives.

The largest single obstacle for using the project systems for analysis is the necessity to understand the underlying practices and the way the support systems are used. This requires validation of the values in fields used by the developers and support technicians to assess the quality and usability of the attribute. Common and serious issues involve missing and, especially, default values that may render an attribute unusable. Any fields that do not have a direct role in the activities performed using the project system are highly suspect and, often provide little value in the analysis. As the systems tend to be highly focused to track issues or versions, extracting reliable data needed for analysis may pose a challenge.

It is worth noting that analyzing data from software support systems is labor intensive. At least 95% of effort should be expected to involve understanding the practices of tool use, cleaning and validating data, and designing relevant measures. There is no guarantee at the outset of the study that the phenomena of interest would be extractable with sufficient accuracy. All too often, such obstacles lead to temptation to model easily-to-obtain yet irrelevant measures, to study phenomena of no practical significance, and to get fascinated by oddities of the tool generated artifacts.

5 Challenges

The motivation to deploy support system based measurement can come from immediate and relatively straightforward applications in project management, such as dashboards showing MR inflow and outflow that help visualize when the project is getting late or to anticipate the completion date.

Although the information in software support systems represents a vast amount of untapped resources for empirical work, it remains a challenge to create models of key problems in software engineering based on that data and to simplify and streamline the data extraction and validation process. This raises a question of how best to improve version control and problem tracking tools to facilitate measurement and analysis. Unfortunately, it is not simply a question of what attributes to add — many fields in the existing systems are empty or contain noise in cases where they provide no clear value to the system users. Therefore, it is of interest to study what information developers would willingly, easily, and accurately enter in problem tracking and version control systems.

It remains to be seen how best to characterize a single software project based on its software repositories and what validation is necessary for that characterization. What models would be plausible for a single software project?

To confirm findings based on studies of an individual project it may be necessary to investigate a larger collection of similar projects. How to minimize the effort needed to validate the compatibility of practices in such a large sample of software projects?

This leads to questions about the role of software repositories in design, planning, execution, and analysis of experiments. Because the models of projects, people, and code tend to be based on the properties of changes it is of interest to know which properties are the most important. In other words, what is the "sufficient statistic" for a software change? A sufficient statistics is a statistical term meaning a summary information that is needed to know all the relevant properties of a sample. For example, a sample from Gaussian distribution can be summarized with just two numeric values — sample mean and sample variance.

Even though it appears that the use of software repositories should enable answering novel software engineering questions, most of these questions have yet to be identified.

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] D. L. Atkins, A. Mockus, and H. P. Siy. *Value Based Software Engineering*, chapter Quantifying the Value of New Technologies for Software Development, pages 327–344. Springer Verlag Berlin Heidelberg, 2006.

[3] Per Cedeqvist and et al. *CVS Manual*. May be found on: `http://www.cvshome.org/CVS/`.

[4] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Subversion Manual*. May be found on: `http://svnbook.red-bean.com/`.

[5] D. Cubranic and G.C Murphy. Hipikat: A project memory for software development. *TSE*, 31(6), 2005.

[6] T Dinh-Trong and Bieman J.M. Open source software development: A case study of freebsd. *IEEE Transactions of Software Engineering*, 31(6), 2005.

[7] Birgit Geppert, Audris Mockus, and Frank Rößler. Refactoring for changeability: A way to go? In *Metrics 2005: 11th International Symposium on Software Metrics*, Como, September 2005. IEEE CS Press.

[8] Daniel German and Audris Mockus. Automating the measurement of open source projects. In *ICSE '03 Workshop on Open Source Software Engineering*, page Automating the Measurement of Open Source Projects, Portland, Oregon, May 3-10 2003.

[9] 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.

[10] Todd L. Graves and Audris Mockus. Inferring programmer effort from software databases. In *22nd European Meeting of Statisticians and 7th Vilnius Conference on Probability Theory and Mathematical Statistics*, page 334, Vilnius, Lithuania, August 1998.

[11] James Herbsleb and Audris Mockus. Formulation and preliminary test of an empirical theory of coordination in software engineering. In *2003 International Conference on Foundations of Software Engineering*, Helsinki, Finland, October 2003. ACM Press.

[12] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. In *23nd International Conference on Software Engineering*, pages 81–90, Toronto, Canada, May 12-19 2001.

[13] Miryung Kim and David Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *International Workshop on Mining Software Repositories*, 2005.

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

[15] Audris Mockus. Description and roadmap for a system to measure open source projects. Link to roadmap off `http://sourcechange.sourceforge.net/`.

[16] Audris Mockus. Analogy based prediction of work item flow in software projects: a case study. In *2003 International Symposium on Empirical Software Engineering*, pages 110–119, Rome, Italy, October 2003. ACM Press.

[17] Audris Mockus. Empirical estimates of software availability of deployed systems. In *2006 International Symposium on Empirical Software Engineering*, page to appear, Rio de Janeiro, Brazil, September 21-22 2006. ACM Press.

[18] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[19] Audris Mockus and James Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *2002 International Conference on Software Engineering*, pages 503–512, Orlando, Florida, May 19-25 2002. ACM Press.

[20] Audris Mockus and Lawrence 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.

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

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

[23] Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *2003 International Conference on Software Engineering*, pages 274–284, Portland, Oregon, May 3-10 2003. ACM Press.

[24] Audris Mockus, Ping Zhang, and Paul Li. Drivers for customer perceived software quality. In *ICSE 2005*, St Louis, Missouri, May 2005. ACM Press.

[25] Rational Software Corporation. *ClearCase Manual*. May be found on: `http://www.rational.com/`.

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

[27] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions of Software Engineering*, 30(9), 2004.