Mining Software Repositories for Rigorous Empirical Evaluation: A Position Paper

Danhua Shao, Sarfraz Khurshid and Dewayne E Perry
Electrical and Computer Engineering, The University of Texas at Austin
{dshao, khurshid, perry}@ece.utexas.edu

Abstract

In the software tools studies, most of the evaluations were done within artificial contexts at labs. Although this approach can give instant feedback with low cost, the mock contexts are quite different from the real context and industrial developers are still confused with the experiment results. Our study provides a significant and low cost method to evaluate software tools with near real contexts.

1. Introduction

Historical data has long been recognized as a rich source of data for empirical studies [1]. It offers significant advantages for descriptive, relational and experimental studies: availability, low costs, avoidance of a number of internal validity issues, etc. There are, of course, some drawbacks: the data may be incomplete; desired data may be missing; the data may be untrustworthy; they data may not be in a convenient form; etc.

In software engineering, there is a rich history of using historical data for a variety of purposes from project management, process improvement, and system analysis to system improvement. An early example related to project management is that of Boehm's book on software economics [2]. It relies on a wealth of historical data to derive his constructive cost model (COCOMO). A variety of studies on software faults also rely on historical data to help understand the problems with software development and evolution (for example, [3] and [4] by one of the authors[1]) in terms of interface and development software faults.

A significant amount of work based on the historical data from one of the subsystems of AT&T/Lucent Technologies' 5ESS™ Telephone Switching System was done in the Software Production Research Department of Bell Labs. This Project was called the Code Decay Project and was the result of an NSF funded collaboration between this department, the National Institute for Statistical Sciences (NISS) and several universities (for example, Adam Porter at the University of Maryland and Nancy Staudenmeyer at Duke University). The historical data consisted of the entire version management and change management histories of this representative subsystem. The studies ranged from looking for factors for code decay [5] to studies of parallel changes showing a direct linear relationship between degree of change concurrency and the increased presence of faults [6].

2. The ICSE MSR Series

The first workshop on mining software repositories was held in Edinburgh in May 2004 with Ahmed Hassan, Ric Holt and Audris Mockus as the co-chairs of the workshop. This was the first such organized community focus on the use of historical data, specifically version management and associated data.

The genesis of this series was Ahmed Hassan's PhD thesis [7] on the use of software repositories to aid software developers and project managers. The general topics for this first workshop were centered around the following: extracting and presenting data from version and associated repositories; using the data to understand system development and evolution processes; using the data to understand system defects and change patterns; using the data to aid in system comprehension and reuse; and using the data for assisting project management. Indeed, we presented a paper on using the 5ESS™ version and change management to understand the phenomena about and effects of small source code changes [8].

While the call for papers for MSR 2005 acknowledged that "Software practitioners and researchers are beginning to recognize the potential benefit of mining this information to *support the maintenance of software systems*, *improve software design/reuse*, and *empirically validate novel ideas and*

[1] Please note that there is a wealth of citations that could be used here and for all the topics covered. We are not trying to be exhaustive, but merely indicative of the variety of approaches to be found – and as this is a position paper, the sources at hand are the easiest to use as illustrations.

techniques", the papers published and the sessions organized were, in general, similar to those found in MSR 2004. There were in fact no papers explicitly concerned about using or mining software repositories for empirically validating novel ideas and techniques. At best, the repositories were used empirically in a self-referential way to evaluate the techniques and tools for mining software repositories

The MSR 2006 was similar to MSR 2004 and 2005. This is not meant as a criticism, but as an observation. The work done so far in mining software repositories has been extremely good and productive. But the focus is not nearly as broad or as interesting as it might be. Certainly the work that has been done provides a useful basis for a variety of explorations that can use the tools and techniques that have been produced thus far. But there is much more that can be done in mining repositories.

3. Rigorous Empirical Evaluation

As we mentioned in the introduction above, historical data provides a rich source for empirical studies. Further, version and change management repositories are amongst the richest data repositories available to software engineering researchers, either from open source projects or from company specific projects such as the 5ESS™ repository used to advantage by Bell Labs researchers.

It is our position that the version and change repositories offer a significant opportunity for supporting rigorous empirical evaluation and validation. Perhaps the most obvious subject domains for this rigorous empirical validation are those domains that are responsible for finding faults in software systems. Version and change management repositories are a natural basis for empirically evaluating analysis tools. It may well be a useful basis for evaluating testing techniques as well, but our experience using these repositories has been with analysis tools rather than testing.

The idea of using historical data as found in such repositories as the basis for software engineering experiments was first suggested in the Perry et al. ICSE 98 tutorial on empirical studies in software engineering [9]. The third part of the tutorial focused on suggested experimental designs and various approaches that might prove useful in future empirical studies. One of these suggested designs was the use of historical data to experimentally evaluate methods, techniques, processes and tools.

An early example of this approach is found in Atkins et al. [10] (not surprisingly all from the same department at Bell Labs as Perry and Votta with Porter as a long time collaborator) where they evaluated the effect on productivity of a version editor, VE, on the basis of historical version management data. The authors were able to differentiate versions edited using VE versus other editors because VE left identifiable signatures in the source code of all the version oriented lines it generated. Using an effort estimation technique developed by Graves and Mockus [11], they were able to demonstrate that those developers who used VE were "approximately 36% more productive when using VE than when using standard test editors."

The effort estimation technique and the comparative data were all derived from version repositories. It was fortunate that they were able to differentiate the historical data into VE and non-VE related groups. Without that extra-repository distinction (or relation if you will), the evaluation would not have been possible. Thus, while this empirical study did use historical data from version and change management, it was, in a real sense, instrumented data.

But the study does illustrate an important point: the versions found in the repository were separated into two groups based on a criterion that was useful in the empirical study and a significant result was obtained on the basis of this differentiation.

The first step then, in designing an evaluative experiment is establishing the *hypotheses* we want to test in terms of *independent* and *dependent variables*. The critical experimental issue here is that of *construct validity*: do we have the right abstract constructs that represent what we intend to investigate and are they represented by useful and appropriate observable constructs.

The independent and dependent variables may include any or all of the following in version and change management databases: versions, changes, dates, people making changes, the kinds of changes, the size of changes, the size of modules, the number of changes, etc. The list of possible variables is limited only by the historical data found in the repositories.

Where data is not found in the repositories, it may be possible to infer it from the data that is there. This is what, for example, Hassan [7] and Mockus and Votta [12] have done in their techniques of determining whether changes are fault fixes, improvements or enhancements (both techniques derived and validated from studying the historical data in these repositories).

The next step is determining the sample groups to be the subjects of the experiment. One should have at least two groups: a control group and a treatment group. Further, one may wish to block on the basis of certain characteristics in the population such as type of change, etc., in which case multiple groups of representative samples will be needed.

On the basis of these desired characteristics, one may then mine the repositories to create equivalent groups made up of either the entire populations in the repositories or randomly selected subsets to use in the experiment.

It is at this point where the usefulness of having both version and change management repositories becomes most interesting in terms of evaluating analysis tools. Given the versions chosen in the subject groups, one can mine the change management repositories for fault fix changes related to those versions and establish the fault set for each version.

This approach provides the "mundane realism" that has not been, and very likely cannot be, provided by fault seeding. Further this removes the *internal validity* problems associated with fault seeding: the representativeness of the faults seeded the placement of those faults, and the frequency of fault occurrence.

Unfortunately, the fault set derived from the change management repository may not be without its own internal validity problem depending on the data therein: faults are often only collected once integration or system testing begins and so the faults found in unit testing are not represented. Thus, to understand and evaluate the behavior of analysis tools for the full range of fault finding, one would have to design and execute field experiments using real development projects to supplement the experiments on historical data and to cover the full range of usefulness for that tool.

4. An Example Experiment

We have done just such an experiment to rigorously evaluate the usefulness, effectiveness and practicality of a tool that detects semantic interference between versions [13]. The idea of this tool grew out of one of the authors work on the problem of parallel changes [6]. There we looked primarily at syntactic interference and hypothesized the existence of significant semantic interference that was not detected by current techniques. The tool is intended to be part of the pre-version-deposit process.

The resulting tool we created is sound but not complete. The experiment was intended to show just where its effectiveness and usefulness lay.

We randomly selected three groups of versions: those where the changes between versions where intervals between versions were quite long; those where the intervals between versions were moderate in length; and those where the intervals between changes were quite short. The three groups were intended to represent versions that were quite stable, versions where the changes could be absorbed and understood by the developers without undue pressure, and versions where the changes were highly "parallel" – i.e., where there is little time to absorb the meaning of the changes and there is significant pressure to hurry changes so the next developer can make his or her changes in a timely way.

We then mined the change repository to create the fault set for these versions, including the date when the fault was found, and further classified the versions according to whether they were fault fixes, improvements or enhancements.

Given the historical data we had mined from the repositories, we then executed the experimental manipulations; we used the tool to analyze the versions in the three groups yielding what we term "direct" semantic interferences (i.e, we do not do pointer analyses).

We then analyzed the fault set to see which of the interferences represented faults and which did not (i.e., which represented either intended interferences – fixing faults – or, possibly, faults that had not been discovered; we assumed that the former was the case). The matched interferences with faults provides a predictive measure for what would be the results of analysis as versions are deposited into the version management system; the unmatched interferences were a measure of false positives that might be encountered in these analyses. We believe, however, that the intended cases of direct semantic interferences are easily dealt with by the developer and represent little overhead in the version deposit process. The fault date data provides a measure of how much time that would be saved in uncovering the faults at deposit time rather than at testing or release time. We further analyzed the fault set to see what kinds of faults were not due to direct semantic interference and classified them as to types of faults.

The full design of the experiment is presented in [14] and the results of executing the experiment are presented in [15]. Our results showed where the tool

was effective (i.e., for which group and which type of versions), how effective it was, and the costs and savings of using it. The results were congruent with the earlier findings in with respect to the degree of concurrency and the likelihood of faults [6].

5. Conclusions

It is time to go beyond the simple demonstrations of ideas that we currently find in research papers, showing with a few artificial cases that something does indeed work. It is time to provide rigorous empirical evaluations and validations that provide useful and practical information about where the tool use is appropriate, the extent to which it is effective, and the practical benefits to be obtained from its use.

The use of version and change management repositories provides a rich and effective basis for doing those rigorous experiments. Further, the methods, techniques, process and tools that have been developed to understand and manipulate these repositories (and presented in the MSR workshop series) provide a rich set of tools to help in these experiments.

We can avoid the artificiality of many of the approaches now used to provide ineffective and shallow evaluations by using industrial strength data from these repositories by carefully constructing appropriate sample populations of versions and faults and their related data. We can provide the needed "mundane reality" [1] that is needed to provide at least one important element in *external validity* – namely, that the experiment was executed in the context of real development data.

Thus, it is time to build on the first generation of mining software repositories and begin the second generation where we use the results of the first to leverage those of the second. This second generation should focus on using these repositories for rigorous empirical evaluation and validation of "*novel ideas and techniques*".

6. References

[1] Rosenthal and Rosnow, *Essentials of Behavioral Research: Methods and Data Analysis*. Second edition. McGraw Hill (Series in Psychology) 1981, 1994.

[2] Barry Boehm, *Software Engineering Economics* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[3] Dewayne E. Perry and W. Michael Evangelist. "An Empirical Study of Software Interface Errors", *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, 32-38.

[4] Dewayne E. Perry and Carol S.Steig, "Software Faults in Evolving a Large, Real-Time System: a Case Study", *4th European Software Engineering Conference -- ESEC93*, Garmisch, Germany, September 1993.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data", *IEEE Transactions of Software Engineering*, TSE 27-1 (January 2001), 1-12,

[6] D.E. Perry, H.P. Siy, and L.G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study", *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 3, July, 2001, pp 308-337.

[7] Ahmed E. Hassan, "Mining Software Repositories to Assist Developers and Support Managers , *PhD Thesis*, School of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 2004.

[8] Ranjith Purushothaman and Dewayne E Perry, ``Towards Understanding the Rhetoric of Small Source Code Changes" Special Issue on Mining Software Repositories, *IEEE Transactions on Software Engineering* TSE 31-6 (June 2005)

[9] Dewayne E. Perry, Adam P. Porter and Lawrence G. Votta, "Tutorial: A Primer on Empirical Studies", *1997 International Conference on Software Engineering*, Boston Mass, May 1997.

[10] David Atkins, Thomas Ball, Todd Graves and Audris Mockus. "Using Version Control Data to Evaluate the Impact of Software Tools", *21st International Conference on Software Engineering*, May 1999, Los Angeles CA.

[11] T. L. Graves and A. Mockus. "Inferring change effort from configuration management data", *Metrics 98: Fifth International Symposium on Software Metrics*, November 1998, Bethesda MD, 267-273.

[12] Audris Mockus, Lawrence G. Votta: Identifying Reasons for Software Changes using Historic Databases. *International Conference on Software Maintenance 2000*, 120-130.

[13] G. Lorenzo Thione and Dewayne E. Perry. "Parallel Changes: Detecting Semantic Interferences". *The 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, July 2005

[14] Danhua Shao, Sarfraz Khurshid and Dewayne E. Perry. "Mining Change and Version Management Histories to Evaluate an Analysis Tool: Extended Abstract", *Mid-Atlantic Student Workshop on Programming Languages and Systems*, April 2006. New Brunsiwck NJ., April 2006

[15] Danhua Shao, Sarfraz Khurshid and Dewayne E Perry. "Detecting Semantic Interference in Parallel Changes: An Exploratory Case Study". Submitted for publication.