Electronic Communications of the EASST
Volume XXX (2014)

Proceedings of the 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014)

Efficacy Measurement of Early Intervention Techniques

Dave Donaghy and Tom Crick

3 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Efficacy Measurement of Early Intervention Techniques

Dave Donaghy[1] **and Tom Crick**[2]

[1] dave.donaghy@hp.com
HP Bristol, UK

[2] tcrick@cardiffmet.ac.uk
Department of Computing
Cardiff Metropolitan University, UK

Abstract: Compiler technology has, for some considerable time, been sufficiently advanced that individual programmers are able to produce, in reasonably short periods of time, tools that might aid with the development process in novel ways: for example, one can easily produce a C compiler tool that will detect uncommon uses of integer arithmetic (such as the rare multiplication of values that are commonly only added) and flag such uses as potential errors.

However, there is currently no convenient way to measure the efficacy of such techniques: where one might *assume* that uncommon uses of integer arithmetic *might* be erroneous, we do not have a way of measuring the cost saving associated with the potential early detection of occurrences of such things.

We present a method of measuring the efficacy of a single *early intervention*, based on the replaying of previous executions of a compile-build-test cycle. This measurement process allows us to identify the software errors that were introduced during an original development and subsequently fixed; additionally, it allows us to identify the subset of such errors that would have been identified by the early intervention. By these means, we can take an existing historical record of a development, and extract from it meaningful information about the value of a proposed new early intervention technique.

Keywords: Verification, Software Engineering, Efficiency, Version Control Systems, Repository Mining

1 Introduction

It is possible for software developers to utilise all manner of tools that will analyse practically any aspect of their source code and report on it in any way they choose; while this freedom will allow arbitrary invention on the part of the software developer, it might also allow construction of analysis tools that *seem* effective, but in reality are not; additionally, individual developers might have different ideas of what constitutes an effective tool [ZVDv08]. It would be useful, therefore, to formalise two separate ideas:

1. What constitutes effectiveness in the realm of compilation tools?
2. Is my tool effective?

2 Robust Efficacy Measurement

Some notions of tool effectiveness during code development and compilation might be as follows:

1. Will my tool make my code better?

2. Since the beginning of my current project, how many errors would have been detected using my tool, which was not in use (or in fact conceived) at the time?

Clearly these two criteria have been written deliberately to highlight the difference between objective, measurable criteria and subjective, hard-to-measure ones. We thus propose (and suggest ways for answering) the following question: *how can we phrase objective questions about measuring the effectiveness of tools run at the time of software compilation?*

3 Retrospective Measurements

We often have access to massive amounts of historical data in the form of a source-code repository [BKPS97, ZZWD05], such as those used by Subversion or Git (among many other tools used for such purposes). Often, though, while these repositories provide the ability to take time-based snapshots of workspaces that "work" (in some loose sense) at a given point in time, we may not take full advantage of the information in them [BRB+09, SK12].

Imagine that we have access to a new compiler option that will allow us to simply prevent compilation of code where a certain kind of error is detected. Assuming we have access to full historical information in our source-code repository, perhaps from repository mining, we can simply re-run our compile-test cycle with the new tool in place, and see which errors would have been identified before their actual detection time. Again, with full historical information, we can identify *how much earlier* each would have been identified, and then make judgments about the benefits of the new tool.

The remaining question, then, is this: if we *do not* have full historical information (and it is likely that we do not), then how much information do we need to judge the efficacy of our new techniques?

4 Open Questions

The following questions may be answered by constructing suitable experiments with data from historical repositories:

1. What build system changes are necessary to allow the retrospective addition of compiler tools to the build cycle?

2. What artefacts must be stored in a repository in order to reproduce compile-build test cycles and identify all errors?

3. How often must snapshots be taken in order to identify the points in time where errors were identified and fixed?

4. What is the potential for false alarms raised by error-checking tools in this proposed system?

Bibliography

[BKPS97] T. Ball, J.-M. Kim, A. A. Porter, H. P. Siy. If Your Version Control System Could Talk... In *Proceedings of the ICSE'97 Workshop on Process Modeling and Empirical Studies of Software Evolution*. 1997.

[BRB+09] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, P. Devanbu. The promises and perils of mining git. Pp. 1–10. IEEE Press, 2009.

[SK12] B. Sisman, A. Kak. Incorporating version histories in Information Retrieval based bug localization. In *Proceedings of the 9th IEEE International Working Conference on Mining Software Repositories (MSR'12)*. Pp. 50–59. IEEE Press, 2012.

[ZVDv08] A. Zaidman, B. Van Rompaey, S. Demeyer, A. van Deursen. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *Proceedings of the 1st International Conference on Software Testing, Verification and Validation*. Pp. 220–229. IEEE Press, 2008.

[ZZWD05] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31(6):429–445, 2005.