CS 687 Empirical Software Engineering Project
Fall 2009

IN PROGRESS - NOT YET COMPLETE!!!!!!

Second UK Workshop on Experimental Software Engineering

Program Chair: Jane Hayes

Program Committee: CS 687 students

The University of Kentucky Workshop on Experimental Software Engineering provides a forum for discussing current experimental studies in the field of software engineering. Papers are solicited for the studies listed in this CFP, as well as for other studies. Accepted papers will not be published in any conference proceedings. Submitted papers must not have been published previously, but they may be submitted elsewhere in the future. All submitted papers will be accepted.

Full-Length Papers: Papers should be submitted 1.5 or double-spaced in a font size no smaller than 11 points, fully justified. Papers must not exceed 25 double-spaced pages including references and figures, and will not be refereed by external reviewers. All papers should indicate what is interesting about the presented work. The first page should include an abstract of maximum 150 words, a list of keywords, and the complete address (including phone and e-mail address) of the author. The citations and references should be formatted in standard software engineering format, that is, with bracketed citations ("[1]") and citation keys that are either numeric or strings based on the authors' names ("[Basi91]").

Artifact Submission: All software artifacts that you use must be submitted to the ~~SEEWeb experimental software repository and the~~ PROMISE software engineering repository.

Presentations: You will be allowed 25 minutes for your presentation, including 5 minutes for questions – **this is subject to change based on number of students.**

Submission Procedure: Three hard copies of a first draft of each paper must be submitted before 27 October to Program Chair J. Hayes (unless conference paper system is up). Each paper will receive at least three reviews, one from the program chair and two from program committee members. Reviews will be returned on 3 November, and the final paper must be submitted electronically by 17 November. Final papers must be submitted in either PDF or DOC format. The final paper must be single spaced and in 10 point font (~10 pages – suggest that you use this format: http://conferences.cis.unisa.edu.au/2006/tabletop2006/IEEE/Format/instruct.htm).

Milestones	Date
Topic selection:	15~~8~~ September
Experimental design review:	29~~2~~ September

Draft paper submitted:	~~27 October~~3 November
Reviews due:	10~~3~~ November
Final paper submitted:	24~~17~~ November
Presentations:	24~~17~~ November to end of class

SUGGESTED TOPICS LIST – this is very preliminary and will be updated

Following is a list of suggested topics for your empirical study. You may choose any topic you wish, either from this list or something of your own creation. I specifically encourage you to consider carrying out an experiment related to your current research.

You will notice that most of these studies do not involve much if any programming but some will involve a lot of program execution. Also, these studies can be done more easily with clever use of shell scripts. There can be a fair amount of overlap between these studies, and you may want to share programs, test data sets, or other artifacts. Trading of this kind of experimental artifacts is greatly encouraged.

Some of these studies could use a partner to carry out some of the work, so as to avoid bias from having one person conduct the entire experiment. I encourage you to help each other; please communicate among yourselves if you need help ... ask and offer.

These descriptions are concise overviews ... I will be available to discuss each project individually during office hours and through email.

Empirical Studies Suggestions – from Jeff Offutt

1. *Java mutation experiments:* One resource we have available is a mutation testing system for Java, mujava <http://ise.gmu.edu/%7Eoffutt/mujava/>. Instructions for downloading, installing, and running mujava are available on the website. There are several small experiments you could use mujava to run.
    * Test criterion comparison. For a collection of programs, develop tests that kill all mutants, and develop tests that satisfy another criterion (data flow, MCDC, edge-pair, input parameter modeling, etc.). Compare them on the basis of number of tests and on their fault finding abilities.
    * Mutation operator evaluation. One key to mutation testing is how good the operators are. Most of the class-level mutation operators are fairly new, and it is possible that some are redundant and others have very little ability to detect faults. It would be helpful to have an experiment to

evaluate the operators, based on their abilities to find faults, redundancy, or frequency of equivalence.
  * Mutation as a fault seeding tool. One use of mutation is to create faults for other purposes, for example, to compare other testing techniques.

2. Web Modeling and Testing Evaluation: I have recently proposed a method for modeling the presentation layer of web applications. This model can be used to generate tests, among other things. If you have access to a reasonably sized web application, it would be very interesting to apply this test method to evaluate its effectiveness. A draft paper is available upon request. **– note from Dr. Hayes (please see me and I will request the paper from Dr. Offutt)**
3. *Software Engineering Factoids:* We have a lot of truisms about software engineering. These are small facts, or "factoids" that "everybody knows" is true, yet the source for these factoids are lost in the mists of time. Some are based on data from the 1970s, some are based on 30 year old casual observations, and some were probably made up by speakers who wished for a fact to support some point. By now, "everybody" accepts these factoids as truth, yet they may no longer be true or may have never been true! A few example factoids are:
    * 80% of bugs are in 20% of the code.
    * 60% of maintenance is perfective, 20% is adaptive, and 20% is additive.
    * 10% of programmers are 10 times more productive than the other 90%.
    * Software is 2/3 maintenance, and 1/3 development.
    * 90% of software is never used.
    * The number of parameters to subroutines is always small.
    * Object-oriented software is less efficient.

   I am sure that you can think of more. The goal of this project would be to verify one or more of the factoids. This would require three steps: (1) find the old sources for the factoid, who originated it, what the fact was based on, and where it was used; (2) verify whether the factoid is true for current systems; and (3) quantify the correct version of the factoid as best as you can from current data.

4. *Metrics Comparison:* Researchers have suggested a large number of ways to measure the complexity and/or quality of software. These software metrics are difficult to evaluate, particularly on an analytical basis. A interesting project would be to take two or more metrics, measure a number of software systems, and compare the measurements in an objective way. The difficult part of this study would be the evaluation method: How can we compare different

software metrics? To come up with a sensible answer to this question, start with a deeper question: What do we want from our metrics?

5. Frequency of Infeasible Subpaths in Testing: Many structural testing criteria exhibit what is called the /feasible path problem/, which says that some of the test requirements are infeasible in the sense that the semantics of the program imply that no test case satisfies the test requirements. Equivalent mutants, unreachable statements in path testing techniques, and infeasible DU-pairs in data flow testing are all instances of the feasible path problem. For example, in branch testing, one branch might be executed if /(X = 0)/ and another if /(X != 0)/; if the test requirements need both branches to be taken during the same execution, the requirement is /infeasible/. This study would determine, for a sample of programs, how many subpaths that are required to be executed by some test criterion are infeasible. A reference on the subject of the feasible path problem can be found on my web site: _Automatically Detecting Equivalent Mutants and Infeasible Paths_ <http://ise.gmu.edu/faculty/ofut/rsrch/abstracts/cbt-equiv.html>.

6. *Traceability experiments:* Much work is being done in the tracing of textual artifacts in an automated manner in the SVV Lab at UK. A number of experiments could be run, such as: how well does the tracing tool RETRO perform on source code or other structured artifacts? If the textual information from a graphical artifact such as a UML diagram was entered into RETRO, how well does RETRO perform on tracing?
Senthil Sundaram, a PhD student in the SVV Lab, has some ideas for experimentation in this area. Please contact him for additional information.
What are some ways to measure whether or not a high level requirement has been satisfied by its children elements? Does one method work better than another? Or with less bias? Ashlee Holbrook, a PhD student in the SVV Lab, has some ideas for experimentation in this area. Please contact her for additional information. – this is outdated…. Wei-Keat Kong and Jody Larsen are working in this area now – Ashlee and Senthil now go by "Dr."!

7. Empirical Experiments in Re-Factoring and Maintainability: Several models for maintainability have been developed in the SVV lab. Some data is available for building or validating models for things such as: estimating the number of changes that a class will require, estimating the effort needed to change a class, etc. Liming Zhao, a PhD student in the SVV Lab, has some ideas for experimentation in this area. Please contact him for additional information at lzhao2@uky.edu.

8. Dr. Dekhtyar has some ideas for a study that looks at how analysts work with traceability tool output. This study requires getting at least two groups of people to look over traceability results. He will give me more details next week.