This article is part of the IEEE Reliability Society 2008 Annual Technology Report.

Some Progress in Software Testing Technology

Phillip A. Laplante
Email: plaplante@gv.psu.edu

Robert Bucholz, and Albert Elcock

Several important applied software testing methodologies have been developed and validated through experimentation by our research group. For each of these technologies we describe the current status of the projects, usable results, limitations of the work and future research and experimentation possibilities.

Estimating total software defects using capture-recapture models

Used by software engineers since the 1970s to estimate the number of defects in software code, the capture-recapture method was first proposed by Laplace in 1786 for biological populations. The model can be described by the following scenario. Suppose a ranger wishes to estimate the number of wolves in his park. He captures, tags, and releases as many wolves as possible during a fixed period of time called a "capture event". After the event the wolves are released and they redistribute throughout the park. The ranger then conducts a second capture event. Suppose that n_1 and n_2 wolves are captured during the first and second events respectively, and m of those wolves are common to both capture events (as determined by the tags). Then the total number of wolves, N, can be estimated using the Lincoln-Peterson Estimator [1],

$$N = n_1 \cdot n_2 / m \qquad (1)$$

Of course, in between the first and second capture events wolves are born and die so that the population is never constant. But N still represents a reasonable estimator of population.

In 1972 Harlan Mills proposed deliberately planting n_1 defects in software under test, conducting a code inspection (which uncovers n_2 defects), identifying the m duplicates, and then using equation (1) to predict the total number of defects present [2]. Instead of seeding deliberate errors, Eick proposed using two inspectors to inspect the same code (finding n_1 and n_2 defects, respectively), identifying the m common defects found, and then estimating total defects using equation (1) [3].

A problem with both of these approaches, however, is that different inspectors do not have identical abilities in finding defects, which might cause one type of defect to be over-represented, and another under-represented. Further, once the software has entered the post-inspection lifecycle, it may be impractical to continuously re-inspect the software after each change.

We have been experimenting with an alternative capture-recapture model employing user reported defects entered into a bug repository to estimate the total number of defects that exist in the software. By harvesting user reported defects dynamic estimates of software defects contained in each release can be made. We have validated this approach with several mature open-source software projects.

Our methodology utilizes the errors reported by users into a defect database or bug repository. To obtain an estimate the first n_1 unduplicated defects are tagged, representing the first capture event. The next n_2 defects reported in the repository represent the second capture event. The m duplicate entries between the two events are then identified. Then the total number of defects in the

software is estimated using equation (1). Because we do not have the limitations of capturing wild animals over some fixed period of time, it is convenient to set the size of the first and second capture events (the number of defects counted) to be identical.

But what is an appropriate size for the ideal capture and recapture event? We have found that the ideal population size is based on a defect's probability of having a duplicate reported within the same release. For the open source defect management system Bugzilla project, we found the ideal capture size to be between 30 and 40 unique defects. This situation can be seen in Table 1.

Table 1 Capture-recapture populations constrained between 30 and 40 defects. Data was collected for several releases of the Bugzilla.

Release	Capture event size	Number of duplicates found	Predicted Defects	Total Defects Reported	Estimation Error
2.14	40	7	228.5	215	0.06
2.16	30	6	150	199	0.25
2.18	35	4	306.25	256	0.19
2.19	30	3	300	256	0.17
2.20	35	2	612.5	557	0.09
2.22	30	5	180	151	0.19

We validated this technique and obtained similar results on several other open source and a closed source projects.

Our capture-recapture model for open and closed source bug repositories has some limitations. These include:

- A sufficient number of defects need to be reported in both the first and second capture events and there must be duplicate defects. More work is needed to determine appropriate necessary and sufficient conditions.

- The predictive model does not take into account defect severity. Additional research and experimentation is needed to see if the technique will work with defects sorted by severity (in essence estimating the population of wolves, rabbits, foxes, squirrels, etc.)

- Certain defects are more likely to appear as duplicates because they are critical or annoying. The model's sensitivity to these factors needs to be further investigated.

- Since some errors are easier to find than others; what is the effect on predictive capability?

- Data collection was manual and difficult. Users notoriously are inconsistent in the way they report errors, making the identification of duplicates (which is crucial to our technique) very challenging. This problem may be mitigated with automated tools, and we have begun to build and test such a tool. But more work is needed to prefect these tools.

Finally, our validation experiments were conducted on a limited number of projects. This promising technique needs to be tested against more and varied project types.

Testing without requirements

Open source software is software that is available in public repositories for use by anyone provided that the provisions of a license are followed. Open source software is increasingly being used in industry and even critical infrastructure systems both to build software and embedded within the software. However, there is little evidence that most open source software is tested, in the traditional sense, against a set of rigorous requirements. In fact, for most open source software no requirements specification exists. How then, can one verify this software rigorously? The solution is to create a set of "behavioral requirements " using available artifacts to document implemented product features as well as expected behavior. The behavioral requirements specification, which looks much like a standard software requirements specification, is used as a basis for traditional software testing.

Since 2006, more than 85 open source projects have been tested using a methodology developed by Elcock and Laplante [4]. In fact, in many cases significant errors were found, reported, and confirmed by open source communities. Because our technique reverse engineers software specifications it also can be used with close source (commercial) software when the existing software specification is known to be incomplete, out-of-version, or incorrect in some way.

The approach to constructing the behavioral specification is a deductive one based on available information for the software application. This information is often found in open source repositories including:

- software requirements or software design documents, even if incomplete, out-of-version, or known to be incorrect,
- user manuals ,
- help files,
- release notes,
- bug reports ,
- support requests,
- application forums,
- experimentation with and use of the application under test,
- ephemera at the application's site or elsewhere.

Having collected and organized all available information and artifacts, best practices are then used to write the behavioral specification. Guidelines found in IEEE Standard 830-1998, *Recommended Practice for Software Requirements Specifications* can be used, for example. The system is then tested using the behavioral specification and traditional testing approaches.

When testing the software using the behavioral specification it can be expected that some test cases will fail. The problem then is determining whether the test case has been incorrectly implemented, a true error has been uncovered, or if there is a defect in the reconstituted specification. Resolving this situation requires repeated retesting, re-examination of the specification, and meticulous documentation [4].

While this methodology has been validated on many small open source projects further validation is required with larger projects, legacy systems, and tools are needed to automate the generation of the behavioral specification document.

Bug fix assignment

Another challenging problem, which has received little attention, is that of assigning reported errors to maintenance engineers for repair according to an appropriate scheduling algorithm. The

order in which errors are fixed matters since users, managers, maintenance engineers, and project sponsors all may have different priorities and resource constraints. Several different repair assignment policies can be used along with appropriate reward mechanisms including:

- first come, first served (FCFS),
- priority fixing (highest priority bugs are fixed first),
- separation of concerns (bugs are clustered and repaired according to functionality),
- random assignment,
- clustering by related problem (errors are grouped together based on their behavior),
- effort assignment**.** (more difficult or easier problems are dealt with first),
- Intelligent assignment (using any of several possible artificial intelligence schemes).

In addition, the behavior of maintenance engineers can be affected by the reward mechanisms that are used. In a true case, for example, software maintenance engineer bonuses depended on reducing the mean time from "new" to "closed" for open problem reports for certain types of reported errors. Unfortunately, bugs could be "closed" by relabeling them as "user misunderstanding," "to be fixed in next release," or by downgrading the seriousness of the bug. The effect was that while bonuses increased, customer satisfaction did not [5].

We used simulations to observe the effects of assignment policy on error backlog and repair time to help inform our understanding of resultant behavior on the part of managers and maintenance engineers. Table 2 summarizes some relevant issues in bug repair assignment policy, management goal, possible metrics to be used, and potential negative behaviors that may emerge on the part of maintenance engineers.

Table 2: Management quality goal, suggested metric(s) and emergent negative maintainer behavior(s) for various error resolution policies [5].

Policy	Management Goal	Suggested metric(s)	Possible negative maintainer behavior (s)
FCFS	Fairness	• Number of errors resolved per time period	• Modification of time stamps • Queue jumping • Forced "hurry up" periods • Careless rushing
Priority	"Important" errors get fixed sooner	• Some linear combination of errors resolved and weights for each priority class	• Downgrading of serious errors • Upgrading of less serious errors • Customer coercion
Separation of concerns	Right person(s) to resolve each error	• Number of errors resolved per individual or group	• Unhealthy competition • Turf battles
Random	Not recommended	• None suggested	• None Suggested
Clustering	Optimization of focus	• Median time to resolve	• Careless rushing
Effort	Controlling maintainer resource allocation	• Median time to resolve • Effort based	• Careless rushing • Effort mis-estimation
Intelligent	None suggested	• Dependent on assignment logic	• None suggested

One area that has received some attention is the use of automated algorithms with machine learning to make repair assignments. In any case, more studies with respect to the appropriate criteria for selecting assignment policy, reward mechanisms and management goals need to be undertaken.

References

[1] G. Seber, The Estimation of Animal Abundance and Related Parameters, Second edition., London, Charles Griffin & Company Ltd., 1982.

[2] H. Mills, "On the Statistical Validation of Computer Programs," Technical report FSC-72-6015, IBM Federal Systems Division, 1972.

[3] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. A. Vander Wiel, "Estimating Software Fault Content Before Coding," Proceedings of the 14th International Conference on Software Engineering, 1992, pp. 59–65.

[4] A. Elcock and Phillip A. Laplante, "Testing Without Requirements," *Innovations in Systems and Software Engineering: A NASA Journal*," vol. 2, December 2006, pp. 137-145.

[5] P. Laplante and N. Ahmad, "Pavlov's Bugs," to appear, IT Professional, December, 2008.

[6] H Petersson, Thomas Thelin, Per Runeson, Claes Wohlin, "Capture–recapture in Software Inspections After 10 Years Research—Theory, Evaluation and Application," The Journal of Systems and Software vol. 72, 2004, pp. 249–264.

[7] G. White, D. Anderson, K. Burnham, and D. Otis, “Capture-Recapture and Removal Methods for Sampling Closed Populations,” Technical report, Los Alamos Nat. Laboratory, 1982.