Using Program Families for Maintenance Experiments

Scott D. Fleming, R. E. Kurt Stirewalt, and Laura K. Dillon
Department of Computer Science and Engineering
Michigan State University, East Lansing, Michigan, USA 48824
{sdf,stire,ldillon}@cse.msu.edu

In general, new modularization techniques require a significant intellectual investment from practitioners in order to adopt them. Before practitioners are willing to make such an investment, they want a careful scientific assessment of the technique for a number of properties (e.g., effects on reusability, reliability, and maintainability). Unfortunately, many promising modularization techniques languish too long in the "academic ghetto," because their validation comes only in the form of simple proof-of-concept examples. To properly assess these techniques, rigorous empirical investigation is needed. Our work is concerned with conducting such empirical investigations for assessing how modularization techniques affect maintainability; in particular, this paper presents an approach to conducting formal experiments for assessing a technique's impact on perfective maintenance. We refer to such experiments as *maintenance experiments* in the sequel.

Fenton and Pfleeger discuss three main types of empirical investigation: *surveys*, *case studies*, and *formal experiments*, which they refer to as research in the large, typical, and small, respectively [4]. We feel that all these types of investigation should be employed in order to properly assess a modularization technique's effects on maintainability; however, when it comes to assessing a new technique, surveys and case studies tend to suffer from the chicken/egg problem. To assess a technique using a survey or case study, practitioners must invest or have previously invested in the technique; but, practitioners require strong evidence of a technique's benefits before they will invest in it (the very reason for doing the study in the first place). It follows that formal experiments are a logical starting point for gathering the evidence needed to convince practitioners to invest in a technique enough to do the other types of investigation.

A formal experiment is a rigorous, replicable investigation of some behavior of interest. Since formal experiments require a high level of control, they tend to be small in scale. In software engineering, formal experiments are commonly used to compare treatments under different conditions. Here, a *treatment* is a tool or a technique [5] (e.g., a CASE tool such as ArgoUML or a method such as RUP). More specifically, an investigator uses a formal experiment to test a *hypothesis*, which is a tentative assumption regarding the effects of the treatments. A formal experiment comprises a series of *trials*, each of which tests a single treatment. In a trial, one or more *experimental subjects* (e.g., software engineering students or professional software developers) apply a treatment to one or more *experimental objects* (e.g., software designs or implementations). Using data gathered from the trials, the investigator analyzes the *dependent variables*, which are the observable factors that varied based on what treatment was applied, while taking into account the *independent variables*, which are the factors that may have influenced how the treatments were applied, to see if the hypothesis was supported or contradicted.

Typically, each trial in a maintenance experiment consists of one or more experimental subjects performing one or more maintenance tasks on a program, called the *base program*, to produce a new program, called the *resultant program* [3, 7]. However, a number of challenges make it difficult to perform trials that accurately simulate maintenance, and are also analyzable and controllable. The first challenge lies in economically obtaining realistically complex base programs, such as multi-threaded base programs with nontrivial synchronization behavior. Different base programs may be needed for different treatments, or to reduce the risk that the selected base programs favor a particular treatment. Another challenge is producing programs that are amenable to reflective comparison when using different treatments. For example, concurrent programs written using different synchronization constructs (e.g., semaphores and monitors) tend to be structured very differently, making them difficult to compare. Comparisons are needed both for interpreting the results of the experiment as well as ensuring that neither program is biased towards a particular treatment. Another challenge is to create base programs that exhibit similar structures to those of "real" programs. The structure of a real program, for example, might reflect the cumulative effects of multiple small changes over time, the effects of multiple maintainers working on the program, and the order that maintenance tasks were applied. A final chal-

First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)
0-7695-2967-4/07 $20.00 © 2007 IEEE

lenge is in supporting replication. Because of the number of variables typically involved with maintenance experiments, extensive replication may be needed to build confidence in the results. For example, Daly et al. performed an experiment on the effects of object-oriented inheritance on maintenance [3], and an early replication by Cartwright found contradictory results [2].

We are investigating an approach using Parnas's program families [6] that should address many of the above challenges. Our approach supports the creation of a regime of maintenance experiments, which are designed to explore a space of related programs. Such a regime compares two or more treatments using a set of program families, one for each treatment, and a set of maintenance tasks. Initially, each family comprises a single program, which we refer to as a *root program*. Our approach iterates over two steps: (1) a maintenance experiment is conducted, which uses equivalent programs from each of the families as base programs, and (2) the resultant programs are added to their associated families. We consider two programs from different families equivalent if they result from the same sequence of maintenance tasks. A notable aspect of our approach is that the program families are populated as experiments are run.

We are currently using our program families-based approach in maintenance experiments to assess a technique we previously invented for modularizing synchronization concerns, called Szumo [1]. For the experiments, we are using two program families: one whose treatment uses Szumo and another whose treatment represents the status quo and uses Pthreads. The root program of each family implements a multi-threaded GUI browser, which reads and displays text from a network server. While the browser is relatively small, it is complex and interesting enough that we previously used it as a pedagogical aid for a software engineering course. We are using three perfective maintenance tasks to populate the program families: one that adds network error handling, another that adds more settings and GUI controls, and another that reads data from multiple servers. Initially, we will conduct our experiments using undergraduate software engineering students. We plan to conduct more experiments using different treatments (e.g., concurrent design patterns) and experts as experimental subjects.

We anticipate several benefits from our approach:

- Reduction in costs associated with creating base programs: Program families are populated with programs resulting from previous experiments. The investigator only needs to create small root programs and to design a set of perfective maintenance tasks that may be applied in various permutations (i.e., to engender branching in the tree).

- Easier reflective comparison of base and resultant programs: Functionally equivalent root programs from different families are easier to compare, because they are small. Functionally equivalent non-root programs from different trees that received the same series of extensions are easier to compare, because they are the result of a series of semantically comparable modifications to their associated roots.

- More accurate simulation of maintenance: Each program (except the root) is the result of some number of maintenance modifications. The program family can be populated such that each extension is made by a different author, and the maintainers are not the same developers that wrote the program. The program family includes all feasible permutations of extensions.

- Supports replication: Program families are easily reused. If the original investigators explore only a portion of the family tree, replicators can continue to populate the program family with new programs from the unexplored parts of the tree. Similarly, a regime can be incrementally populated by replicating the same maintenance experiments but with different treatments.

Our approach raises several interesting research questions. One question is how to select which resultant programs to reuse in subsequent experiments. For example, should the investigator choose explicitly based on some criteria, or should programs be chosen non-deterministically. Another question is what is the best order in which to explore the family trees. For example, does some prescriptive combination of depth and breadth produce better results? A final question is can a cohesive set of program families be conceived such that they form the standard programs for doing maintenance experiments on a particular domain of systems. For example, is our GUI browser program family sufficient for exploring multi-threaded, interactive systems?

References

[1] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of FSE'2000*, 2000.

[2] M. Cartwright and M. Shepperd. An empirical view of inheritance. *Inf. Softw. Technol.*, 40(14), 1998.

[3] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proc. of ICSM'95*, 1995.

[4] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[5] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4), 1995.

[6] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1), 1976.

[7] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6), 2002.

0-7695-2967-4/07 $20.00 © 2007 IEEE