

Deploying Instrumented Software to Assist the Testing Activity

Sebastian Elbaum
Computer Science and Engineering Dept.
University of Nebraska
Lincoln, Nebraska
elbaum@cse.unl.edu

Madeline Hardojo
Computer Science and Engineering Dept.
University of Nebraska
Lincoln, Nebraska
mharidojo@cse.unl.edu

1. Introduction

The idea of leveraging a large user base to drive product improvement is not new. Netscape, Microsoft, and Gnome, for example, have implemented different types of automated failure reports that take a snapshot of the system at the time of the failure to assist in assessment and debugging. On one hand, such approaches seem primitive when compared with existing powerful dynamic analysis techniques. On the other hand, such powerful analysis techniques were developed primarily to assist with the activities conducted in a controlled industrial environment. Such an environment differs from a customer site in at least three aspects.

First, the company environment exposes just a subset of all possible environments and configurations that can be observed in the field. For example, a company developing operating systems can test at best hundreds of possible configurations, even though the total possible combinations might be an order of magnitude larger. Existing analysis approaches must be adapted or new ones created to capture and integrate information from multiple customer sites, providing a richer and more representative sample of how the system is used and how it interacts with its context.

Second, the company environment is likely to be more forgiving than users in terms of software perturbations in the form of unwanted behavior or poor performance. Therefore, instrumenting and monitoring activities on released software must pay additional attention to the concept of transparency so that its behavior is the same as the original or at least appear to be the same to the user and the other pieces of software. Furthermore, new approaches must consider the limitations imposed by other factors such as the network bandwidth and storage considerations that might impact the scalability.

Third, collecting and analyzing information from a customer site raises new issues in terms of privacy and confidentiality. Initial industrial attempts on data collection indicates that this is something that can be overcome given

enough incentives. However, it is not clear how such activities affect the user behavior impacting the usage of the released software.

These differences between the controlled industrial environment and a customer sites offer new opportunities, which have been further empowered by recent technologies that enable remote monitoring and analysis of software application at the customer site. We would like to: (1) know what can be learned by monitoring released software, (2) decide what is worth monitoring, when, and where, (3) determine the cost of obtaining and analyzing such information, (4) find mechanisms to minimize that cost while leveraging the number of observed sites to maintain the potential gains, (5) identify scenarios that are more likely to benefit from this type of approach, and (6) understand how to decrease user resistance to this type of monitoring due to privacy and security concerns. This paper presents specific instances of these questions within the testing arena and introduces our empirical attempt to answer them.

2 Adaptive Testing

We conjecture that testing techniques are limited by their lack of connection with how the software is used in the field, with shifts in user behavior, and with new environments in which the released software is executing. For example, techniques are oblivious to the fact that most test cases belong to configurations rarely present in the field, tests are developed without enough consideration for their level of similarity to user operations, and resources are prioritized without a reassessment for how critical a feature is to the customer.

This situation has negative consequences. First, the testing suite can misutilize the testing resources and generate an uninformed and likely biased assessment of the software quality. Second, testing techniques unaware of changed assumptions can harm the fault detection capabilities of an existing suite. Third, software engineers are forced to make time consuming and difficult, uninformed assump-

tions about the expected behavior of released software. The accuracy of software engineers predictions can have a major impact in the effectiveness and efficiency of the testing activity. These circumstances are likely to result in decreased software quality and reliability over the systems lifetime.

It follows that there is an enormous opportunity to continually support the quality and reliability of evolving software by adapting the testing process and its outcome through the incorporation of information from released software. Recent studies have begun to investigate some of the relevant issues. For example, the perpetual testing project has studied the overhead of leaving residual instrumentation in released software [2], the gamma system has been used to evaluate the efficiency of distributed instrumentation schemes and to simulate their impact on coverage [1], and the dynaInst group has studied the dynamic manipulation of instrumentation to efficiently support various types of coverage monitoring [3]. Still, we have not fully addressed the following questions:

- Can we effectively increase coverage by observing released software? Previous simulation studies have provided an initial estimate for gains, but it is still uncertain how much we can learn from a truly deployed system. We would also like to estimate the effort associated with translating the collected information into a test suite with greater fault detection capabilities. Last, we will like to evaluate how one aspect of the system setting, the system initial test suite, affects the effectiveness of this approach.
- What type of information should we consider: coverage at the block, function, or component level? How should we include configuration and environment attributes? We need to study the tradeoffs between the granularity of the information, its collection and analysis overhead, the accuracy of the integrated information, and the overall effectiveness of the outcome. Furthermore, it is likely that we will have to learn how to combine all those sources of information.
- How much analysis should be performed at the customer site and how much in-house? What are the triggers for sending data for analysis? How often is data transferred to the central repository? There is a cost associated with the analysis activity and a cost associated with the transfer and storage of collected information. Transfer costs have been largely ignored in past studies, but they can become significant as more detailed information is requested. We will study the use of simple anomaly predicates at the customer site to minimize data transfer (e.g., only transfer when a new component is covered or a new module is loaded).
- What is the relationship between the number of users

and the capture of previously unobserved behavior? Is there an asymptotic relationship between the number of monitored sites and the gains? What is the growth rate for bandwidth and storage as the number of monitored sites increases? If we can understand how the number of monitored sites and our system knowledge are related, we might be able to just target the sites that are most likely to increase our knowledge. Learning about the requirements will also prepare us to discuss further scalability issues.

- How likely and drastic are operational profile variations over different periods of time? How does software evolution affect our assumptions about customer behavior in terms of operational profile? Through these questions we want to determine the potential application of the collected information for operational profile refinement. Operational profiles have been used to quantify system reliability and guide the testing process, so it would also be interesting to explore when and how such approaches are challenged by a varying usage and program changes.
- How reluctant are the users to provide the type of information we are monitoring? Can a simple mechanism that ensures anonymity minimize user resistance? Do customers behave differently when observed? The best techniques will fail if we cannot understand and then minimize user resistance. We will try to determine whether such resistance can be overcome through the right type of incentives and guarantees (e.g., anonymity).

3 Study Design

This section shortly describes the subject program, the experimental setting, and the experiment execution (in progress) and the planned analysis to start addressing those questions.

3.1 Subject Preparation

To perform our study we searched for a subject program that was large enough to be representative of a large population of programs, with available source code so that we could freely instrument it to collect the required information, used extensively within our network domain so that we had an initial user population of good size but still certain level of control on its execution and distribution, used extensively outside our domain so that we could extend the study and overcome threats to external validity, and dealt with an activity or content that was important to the user so that we could evaluate the privacy concerns normally raised for this type of monitoring.

Pine (Program for Internet News and Email), a popular unix program for reading, sending, and managing electronic messages, met all those requirements. In addition, pine offers personalization settings through different configurations and supports tens of platforms. Testing such as system usually faces a problem known as “configuration explosion”, which we are also likely to explore. For the first phase of this study we are using Pine, Unix build, version 4.03, which contains 1399 C functions (approximately 8% of the functions are architecture dependent).

To study what can be learned after released, we needed to model the testing process that normally precedes deployment. We started by deriving requirements from the man pages and user’s manual to generate a first set of test cases for the basic functionality. Given the interactive nature of pine, we implemented the test cases as *expect* scripts. We then used white box testing to complement the initial suite. The final suite has 288 test cases that cover 60% of the functions. In addition, to allow us to measure the effectiveness of the initial test suite and the potential benefits of the data captured after release, we required faults. Such faults were not available with our subject program; thus, to obtain them, we followed a procedure similar to one employed in previous studies of testing techniques to seed faults. This effort resulted in 24 seeded faults.

3.2 Settings

In order to collect all the desired data we instrumented the source code for the fault-free version of pine. The instrumentation is meant to capture functional coverage information, operational frequency, changes in the configuration files, and accesses to changed environmental variables. In addition, we incorporate a trigger function that determines at what point the collected data is transferred to the central repository. The default implementation marshals, time stamps, and encrypts the collected information after every user session and sends it to the central repository.

The study began with the identification of five “friendly” users that would adopt our version of pine. The first two weeks served to inspect the data collection process and determine whether the distributed version of pine was self-installing correctly in each account, operating properly, sending the appropriate data to the repository, and removing itself when indicated. After refining the data collection and fixing some faults in the installation and de-installation process, we proceeded to expand the sample of users. At this time, pine has been distributed to approximately 30 active users within our network domain.

3.3 Pending Analysis

After this initial data collection stage, we will be able to answer some of the posted questions. For example,

we will be able to quantify how much coverage can be gained as user participation increases, how true are our pre-release estimates of the operational profiles, what are the repository storage requirements under a full instrumentation, or whether our one-way encryption mechanism to ensure anonymity makes the users any more comfortable about sharing their data.

However, several questions will remain unanswered. For example, what would have happened under a distributed instrumentation scheme where each instance is meant to collect different data?, or how does the quality of the original test suite affect the coverage gains? We are collecting extensive data in this first phase to enable future experiments to answer those questions. For example, since we are employing full instrumentation to collect coverage information, a future experiment could manipulate the initial test suite quality by removing or adding test cases and re-estimating the coverage gains by recounting the data of the collected sessions. Or, since we are attaching an identifier to each user session, and a time stamp to all collected events, we can set an accurate simulation to study what distributed instrumentation scheme would be more efficient while taking into consideration some of the latencies involved.

4 Final Remarks

The study we are conducting on pine is still in progress and evolving. However, preliminary results of this first phase will be available to share with the workshop participants. We are planning a second phase that will include multiple versions of pine and users from various network domains which implies that we will have to incorporate more light weight and focused instrumentation. We would like to invite the workshop participants to suggest further questions and additional means to leverage the data we are gathering and the empirical structure we are building.

Acknowledgments

This work was partially supported by the NSF-ITR Program under Award CCR-0080898.

References

- [1] J. Bowring, A. Orso, and M. Harrold. Monitoring deployer software using software tomography. pages 2–9, 2002.
- [2] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proceedings International Conference of Software Engineering*, pages 277–284, May 1999.
- [3] M. Tikir and J. Holligsworth. Efficient instrumentation for code coverage testing. In *Proceedings International Symposium of Software Testing and Analysis*, pages 86–96.