MULTIPLY-DEPLOYED RESIDUAL TESTING AT THE OBJECT LEVEL

Leila Naslavsky Marcio Dias Debra Richardson

School of Information and Computer Science
University of California Irvine
Irvine, CA, USA
{lnaslavs,mdias,djr}@ics.uci.edu

Abstract

Software testing is a proven technique widely used to increase confidence that a software product behaves as expected and to gather system usage information to support evolution. Many structural coverage criteria techniques are used to perform and measure testing activities, but their complete satisfaction is rarely achieved resulting in code release with neglected test obligations. This paper suggests a remedy for this situation – monitor the software after release to measure coverage criteria (either by beta testers or by end users). Our approach explores residual testing in multiple deployments of an application to determine incrementally but continually how well selected coverage criteria are satisfied. We introduce a prototype and demonstrate the value of the approach to real-world applications.

Key Words

Test Coverage Criteria, Residual Testing, Software Monitoring

1. Introduction

Software testing activities must always compromise between accuracy and cost [15]. One can aim at proving system correctness, performing exhaustive testing or covering all the possible execution paths at the expense of infinite effort [16]. This approach, however, is remote from current state-of-the-practice in the software industry. At one end of the situation are the developers who, as a result of time-to-market pressures and cost constraints, release their products without proper testing and consequently have to resolve problems in the field on an as-demanded basis. At the other end are end users who are irritated at having to cope with low quality products and who also demand improvements to existing solutions.

By addressing the inevitable situation that software products are actually released without 100% testing coverage, we can better contribute feasible remedies for real-world situations. This work contributes one such remedy. Our approach allows product release [1] but still keeps track of system execution in the deployed environment; our purpose is to monitor test coverage achieved and other information useful for fixing the product or improving it by consolidating executions in multiple deployments. The approach uses residual testing techniques [4].

This paper is not concerned with defining coverage criteria per se, but rather with exploring our approach to multiply-deployed residual testing. Thus, for our current purpose we have chosen simple object level coverage criteria. These coverage criteria are used, rather than a control-flow based criterion, because deployed applications should be tested at a higher level of abstraction so as to avoid undue performance degradation. In future efforts we intend to explore not only more sophisticated object-oriented test coverage criteria but also developer-defined coverage criteria guided by specific concerns. This will require enhanced monitoring capabilities and specification of events of interest such as those provided by *MonArch* [17].

The rest of this paper is organized as follows. Section 2 elaborates our motivation. Section 3 explains our approach to residual testing at the object level. Section 4 presents our prototype. Section 5 presents a case study. Section 6 differentiates related work. Section 7 provides conclusions and describes future work.

2. Motivation

We are motivated by the fact that products are often released into the market without adequate testing due to time-to-market constraints and resource limitations. We suggest that testing in the deployed environment could provide the developer with useful feedback – from ensuring that coverage of a certain test criterion has increased to insights for valuable system evolution based upon how the system is used. This feedback should be provided during beta testing and after end-user release. During beta testing, testers are charged with testing and evaluating the software, whereas the developers have an

[1] The term release here applies either to the actual release to end-users or release to beta-testers.

opportunity to gather information for improving the product. Beta testing would be more useful if developers were able to focus feedback on what they consider critical rather than merely receiving bug reports, questionnaires and suggestions. When the product is released to end-users, the kind of information to be gathered may change, but feedback remains key to improving software quality. Thus, for the purpose of software evolution, specifically perfective maintenance, we should monitor system usage after release. This approach has been termed residual testing [4] and is part of the perpetual testing paradigm [11].

Once the system is released, there are typically multiple deployed instances being used by multiple users. Consequently, in addition to performing residual testing of each individual instance, the information gathered from multiple instances should be consolidated to obtain even greater insight into how the system is being used. The results of this consolidated information should then be merged with previous test coverage measures to produce new measures of test adequacy.

3. Residual Testing at the Object Level

Control-flow based test criteria – such as node, edge and path coverage – are considered to be useful for testing individual units. The increasing complexity of software systems has required methods at a higher level of abstraction, such as object-oriented modeling. This has then lead to the need for more abstract testing techniques. For testing object-oriented system integration, for instance, coverage criteria that focus on inter-class and intra-class testing have been defined by abstracting method internals [12], [3], [19].

As our approach to residual testing is at the integrated system level, we explore test coverage criteria defined over call-graphs, a higher level of abstraction than conventional coverage criteria based on control flow graphs.

3.1 Call-Graph Based Test Coverage Criteria

A call-graph is a directed graph data structure that models the structure of a system's program units as nodes and the calls from one unit to another as edges. Harrold and Rothermel introduce the term inter-class call graph [13] as part of their family of code-based representations for object-oriented software. In this work, we use a flow-insensitive inter-class call graph that depicts a method from a given class as a node and a message sent from one method to another as an edge. Thus, an edge from one node to another indicates that the destination node may receive a message from the source node and thus the destination method may be invoked sometime during execution of the source method. Figure 1 depicts an example of such graph, where the four methods (nodes) can be located in single or different classes, the dashed lines shows messages sent and return among the methods, and the solid arrows (edges) shows possible messages sent. It is worth noting that there are some essential differences between the control-flow graph and this call graph. The control-flow graph denotes order and flow of execution. For instance, a fork in the control-flow graph indicates a decision and means that either the left edge or the right edge will be executed. In the call graph, on the other hand, a fork does not indicate selection.

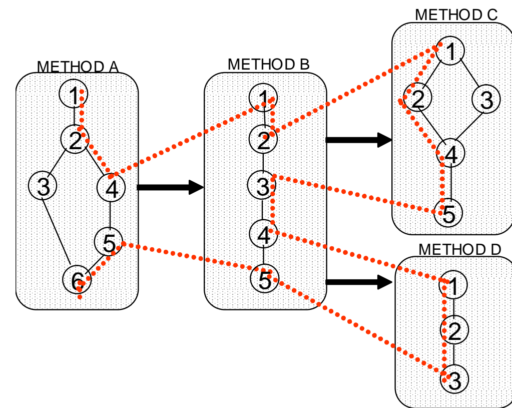

Figure 1 - Flow-insensitive inter-class call graph

The current prototype of our residual testing approach implements a "method execution" coverage criteria, which measures which methods are executed – that is, which nodes in the call-graph that are covered. This object-oriented method is similar to node coverage for control-flow graphs, which measures which statements in a unit have been executed. Our prototype also implements "method-to-method invocation" coverage, which measures which messages between pairs of communicating methods are actually sent. This is similar to edge coverage in control-flow graphs.

In our current prototype, the developer can select specific objects on which to focus the residual testing. The objects may be chosen either because of quality concerns or because they were not sufficiently tested in the development environment. With future enhancements to this work, we plan to provide finer granularity in this selection of focused residual testing.

3.2. Information Gathering for Residual Testing

Gathering information during execution of a deployed application by tracking system usage may reveal defects that must be fixed (corrective maintenance) and also expose possible improvements (perfective maintenance). Further, by monitoring the deployed application, residual

testing according to some coverage criterion can augment test coverage achieved in the development environment prior to product release. So method execution and method-to-method invocation (messages sent between methods) are some of the information that can be captured.

There is a cost imposed, however, by using probes to monitor application execution; in particular, system performance may be diminished through probes. In [4] it is argued that some types of critical applications may never allow any such interference, yet there are other types where this is not a problem, especially during beta-test. Our approach attempts to limit interference with system performance. Although some monitoring is inevitable to perform residual testing, depending on the goals probes are not needed at each and every high-level execution point (method execution and message sent). We increase or decrease occurrence of such probes according to the need for monitoring the coverage criteria selected; this flexibility can reduce impact on system performance.

In addition to coverage information, behavioral information is needed to check the correctness of deployed executions. Thus, residual testing should not only gather information about *structure* that was executed, but also sufficient information about *behavior* to enable comparing results to some test oracle. This comparison can be done in the deployed site or, to avoid interference, in the development environment.

3.3. Consolidation of Residual Testing Results from Multiple Deployments

Information is gathered separately for each execution of the application and each deployed system, thereby providing a set of structural coverage metrics and behavioral results. To obtain an overall view of the residual testing, the data should be consolidated. In so doing, aggregated information about the effectiveness of the residual testing on multiply deployed systems yields more significant system usage profiles and measure of test adequacy than obtained during pre-deployment testing (during development) and also more than could be obtained from a single deployment. For instance, consolidated information may indicate that a given method was executed at least once in each execution or by each deployed system. This consolidation and its respective interpretation provide feedback to the developers about system coverage and use, as well as an insight into adjustments that can be made to the monitoring configuration.

The information gathered is consolidated according to the developer's focus of interest. This interest assumes various dimensions. One dimension would be the deployments; for instance, all data gathered for a single deployed system might be consolidated or all data for all beta installations, etc. Another dimension might be the application features exercised during individual executions, such as testing the graphical user interface or security issues. A third dimension might be the test coverage criterion used. The developer can then choose to consolidate along a single or multiple dimensions. For instance, one consolidation might consider results gathered from all executions of a beta test installation, run by a single user. Another can consider data gathered from all executions on all deployed installations but for a specific usage profile group (e.g. groups that extensively use a specific kind of feature such as GUI, security, or database). Figure 2 illustrates these dimensions of consolidation (although clearly different dimensions are possible).

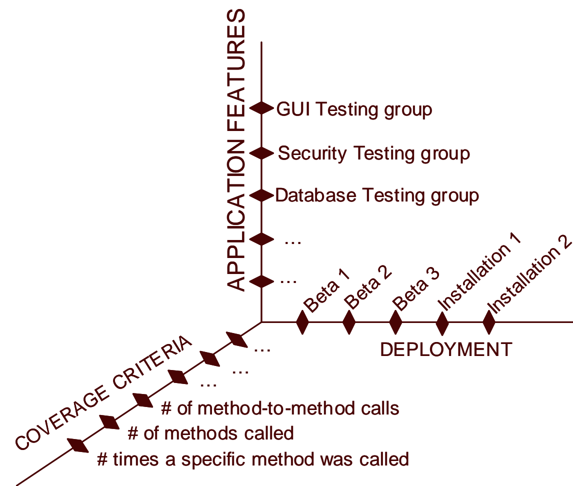

Figure 2 – Possible dimensions of interest.

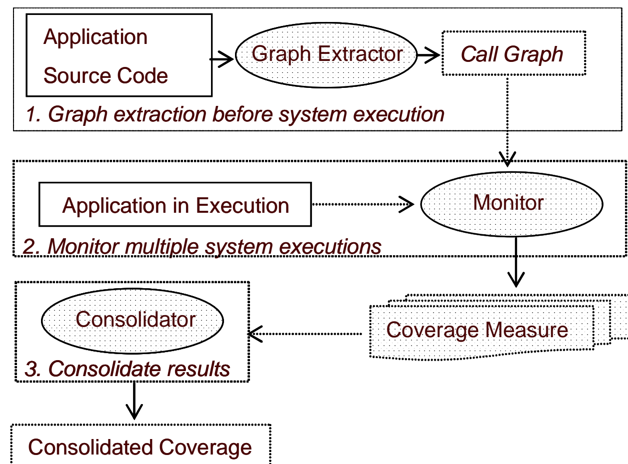

Figure 3 – Prototype Modules.

4. Implementation of a Prototype

To validate our idea we developed a prototype for performing residual testing over Java applications. It is

divided into three modules that reflect the different phases of our approach: The *graph extractor* module develops the static call graph before system execution, the *monitor* module captures data from deployed instance of the application, and the *consolidator* module merges the results obtained from multiple deployed systems, according to the developer's criteria. Figure 3 depicts the modules and how the information is exchanged between them.

4.1. Extracting the Call-Graph

The *graph extractor* is responsible for extracting the call-graph from the application code. In this module, we used the *Recoder* Java framework [20], which provides support for parsing and analyzing Java programs. The call-graph obtained from that analysis has information about existing packages, classes, methods and method references. Additionally, the *graph extractor* supports the call-graph visualization by using ATT's *Graphviz* (*dot*) [8] layout tool for directed graph generation as well as their Java *Grappa* [9] library.

4.2. Capturing Information from Deployed Systems

The *monitor* module uses external probes to capture run-time information in which the developer is interested. We based our solution on the *Java Debug Interface (JDI)* of the *Java Platform Debugger Architecture (JPDA).* The *JDI* is a high-level Java API that supports event collection of Java virtual machines, thus no instrumentation happens to the code. This allows the initial monitoring configuration to be modified whenever needed without the need for changing the application source or the Java intermediate code. This flexibility is handy since one can easily disable (and later enable) probes for part of the application already tested, or enable for the part left to be tested later because it is not initially critical. The list of classes present in the call-graph is the domain for this probes configuration, the developer is allowed to filter the events from certain classes. Nevertheless, due to a granularity limitation, it is not possible to filter the events gathered on a method-by-method basis.

4.3. Consolidating the Captured Information

Structural information is captured separately during execution of each deployed system and must then be merged. The current *consolidator* module provides support for merging results obtained from each separate execution. This flexibility allows the user to select which execution results he wants to consider for a given consolidation. For instance, if a certain beta test installation was executed 10 times, where 5 of those extensively used GUI functionalities and the other 5 extensively used security functionalities, the user can separately consolidate the two 5 executions results subsets and also consolidate all 10 executions results.

Currently, the module shows the total number of executions considered, the total number of methods that were not covered, and the number of methods (nodes) and method-to-method reference (edges) in the call graph. Additionally, it lists the method executed and method-to-method invocation along with the number of executions on which they were covered. To obtain a better picture of each individual execution, the developer can also visualize the call-graph annotated with information on the number of times a method and a method-to-method invocation were executed. This visual information can also be obtained for the consolidation of executions.

5. Case Study

For our case study, we did a single experiment with a single deployment. The application used is the Notepad text editor, available as a *Java Foundation Class* example with the *Java 2 SDK*, Standard Edition. It was modified by adding a *save file* functionality and removing tree document visualization functionality.

5.1. The Notepad Application

This particular application supports the following functionalities: create, open and save a file, move and copy selected text to the clipboard, and paste the clipboard to a selected area, plus undo and redo actions. The application has 1 named class, 11 anonymous classes and 500 lines of code. Additionally, it makes use of the following Java APIs: swing, awt, beans, util and io.

5.2. The Experiment

Once the call-graph was extracted from the source code, we selected only application classes for monitoring (and no Java libraries classes). The monitor captured method entry and exit events from the Java virtual machine running the notepad. We executed the application 10 times, and in each execution we used some of the functionalities. Each of the execution results was saved and later provided as input for the consolidator module.

5.3. Results

The call graph was composed of 44 methods and 38 invocations between those methods. Each of the 10 executions considered covered an average of 37 methods and 33 method-to-method invocations. By merging the executions we were able to cover 41 (93.2 %) of the 44 methods and all 38 (100%) of the invocations. 68.2% of

the methods were executed at least once during each execution; and three of the methods were never executed, although all method-to-method invocations were, which should lead to further investigation of the reasons why and possibly to reengineering the code.

As already mentioned, we modified the application slightly by adding a new *save file* functionality, because the original version did not provide such functionality. The Notepad application is a GUI-intensive application that explores the Java GUI libraries. Some of its methods are invoked from the Java virtual machine and these invocations are not explicit in the code. This adds a level of difficulty to code understanding, which could be eliminated in our case by the call-graph extracted with the *graph extractor* module.

As one example of the type of benefits we were able to identify from the merged results, consider the following. The results showed us that the *setTitle* method (sets the main frame title of the application) was called once when the application was started and once whenever a file was opened (the method *actionPerformed* in the inner class *OpenAction* was called). With this information we realized that if we executed the *open file* functionality followed by the *new file* functionality, the *setTitle* method would not get called and the main frame title would not be cleared. The same happened when we executed the *save file* functionality. So, we could improve our test set to check if the frame title was updated whenever a file changed its name.

6. Conclusions

6.1. Related Work

Residual test coverage monitoring, as introduced in [5], is a technique that allows developers to instrument their code released either to beta testers or to end-users. Their prototype inserts instrumentation to basic blocks in Java class files, and thereby monitors nodes in a program's control flow graph. The process instruments every single block, and the removal of the instrumentation is done automatically based on a cumulative coverage table.

Our approach differs from this work in several ways. First, we monitor at a higher level of abstraction; we are not concerned with basic block executions but rather with method executions and messages exchanged by them. Second, we aim to provide the developer with richer feedback that can be used as the basis for decisions about the monitoring configuration or for maintenance activities. So although the monitoring configuration could be modified automatically by removing the monitoring for the covered methods, our approach allows the developer to specify focus on those classes of interest. Due to a limitation of the technology we used for our current prototype, granularity is at the level of classes. However, we plan to provide future support to monitor methods and even finer granularity.

6.2. Limitations and Future Work

In addition to the limitation with respect to granularity discussed above, we are aware that this work has limitations related to security, privacy and confidentiality s mentioned in Section 3.2 and discussed elsewhere [4]. We are also aware of the interference that probes could introduce in our system, but because they can be inserted or deleted at the will of developers, their interference can be controlled. The metric used in this work is simple and was chosen because the purpose was to experiment with multiple executions.

We plan to continue this work by enriching the information gathered and provided as feedback to the developer by using more sophisticated coverage criteria and more sophisticated monitoring. Thus, in addition to gathering information about the structure executed, we also plan to gather information about the execution's behavior. This kind of information is a key feature for checking the correctness of the executions by comparing the results to system specifications and test oracles.

Clearly, we need to experiment further with our approach to multiply-deployed residual testing. Although the Notepad application is a real-world user application, it is still simple compared to most complex application existing in industry. Nevertheless, having actual users explore the application can lead to information about the system that is closer to reality as was the case of being able to enrich the pre-existing test set. We also plan to experiment with the ideas of consolidation along multiple dimensions, for example including random user profiles, user profile groups testing specific application features, and with multiple, different platforms.

Acknowledgments

The authors would like to acknowledge and thank Henry Muccini and the ROSATEA group in the School of Information and Computer Science at the University of California, Irvine for their valuable insights.

References

[1] A. Memon, M.L. Soffa, M. Pollack, Coverage Criteria for GUI Testing, Proceedings of European Software Engineering Conferece/SIGSOFT, Vienna, Austria, Sept. 2001.

[2] A. Mitchell, J. F. Power, Toward a definition of run-time object-oriented metrics, 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented

Software Engineering, Darmstadt, Germany, July 21-25, 2003.
[3] A. L. Souter, L. L. Pollock, D. Hisley, Inter-class def-use analysis with partial class representations, Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Toulouse, France, 1999
[4] C. Pavlopulou, M.Young, Residual Test Coverage Monitoring, Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States, May 1999.
[5] C. Pavlopulou, Residual Coverage Monitoring of Java Programs, Master's Thesis , Purdue University, 1998.
[6] D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, ACM S1GSOFT Software Engineering Notes 17(4):40-52, October 1992.
[7] G. C. Murphy, D. Notkin, W. G. Griswold, E. S. Lan., An empirical study of static call graph extractors, ACM Transactions on Software Engineering and Methodology (TOSEM) , vol 7(2) ÿ(April 1998)
[8] Graphviz website, www.research.att.com/sw/tools/graphviz/ (accessed on Oct. 01, 2003)
[9] Grappa tool website, www.research.att.com/sw/tools/graphviz/packages/grappa.html (accessed on Oct. 01, 2003)
[10] L. Larsen, M. J. Harrold, Slicing Object-Oriented Software, Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, March 1996.
[11] L.J. Osterweil, L.A. Clarke, D.J. Richardson, and M. Young. Perpetual Testing. Proceedings of the Ninth International Software Quality Week, San Francisco, California,USA,May 1996
[12] M. J. Harrold, G. Rothermel, Performing Data Flow Testing on Classes. Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'94), pp. 154-163, New Orleans, Lousiana, December 1994.
[13] M. J. Harrold, G. Rothermel., A Coherent Family of Analyzable Graph Representations for Object-Oriented Software. Technical Report OSU-CISRC-11/96-TR60, Department of Computer and Information Science, The Ohio State University, November 1996.
[14] M. J. Harrold, J. D. McGregor, K. Fitzpatrick,, Incremental Testing of Object-Oriented Class Inheritance Structures. Proceedings of the 14th International Conference on Software Engineering, Melborne, Australia, May 1992.
[15] M. J. Harrold, Testing: a roadmap, Proceedings of the 22nd International Conference on on Software Engineering, Future of Software Engineering Track, Limerick Ireland, June 2000.
[16] M. Young, R. Taylor, Rethinking the Taxonomy of Fault Detection Techniques, Proceedings of the International Conference on Software Engineering, Pittsburg, PA, USA, May 1989.
[17] MonArch website, www.ics.uci.edu/~mdias/research/MonArch (accessed on Sep. 15, 2003)
[18] P.C. Jorgensen, Software Testing, A Craftsman Approach, CRC Press.
[19] P.C. Jorgensen, C. Erickson, Object-oriented integration testing, Communications of the ACM, v.37 n.9, p30-38, Sept. 1994.
[20] R.V. Binder, Testing Object-Oriented Systems, Models, Patterns and Tools, Addison Wesley.
[21] RECODER website, recoder.sourceforge.net (accessed on Oct. 28, 2003).
[22] S.R. Chidamber, C.F. Keremer, A Metrics suite for Object Oriented Design, IEEE Transactions of Software Engineering, vol 20(6), 1994.