Continuous Self-Evaluation for the Self-Improvement of Software

Lori A. Clarke and Leon J. Osterweil*
Department of Computer Science

University of Massachusetts
Amherst, MA 01003
USA

*This work is being carried out as a collaboration among Leon J. Osterweil, Lori A. Clarke and George Avrunin at the University of Massachusetts, Debra Richardson at the University of California at Irvine, and Michael Young at the University of Oregon.

1.0 INTRODUCTION:

> The purpose of evaluation is to determine the extent to which a software product is meeting its requirements, and to suggest, as much as possible, the sorts of modifications that should be expected to help improve its ability to meet those requirements. Our interest is in improving software. Thus in this paper we propose a self-evaluation approach, called perpetual testing and analysis, that fits nicely into the larger activity of self-improvement.

In a manual, or human-driven, software improvement process humans take the lead in carrying out the testing and evaluation of software, humans infer the changes that need to be made, humans effect the changes, and humans reinstall the modified software, at which point the improvement cycle begins again. Software self-improvement suggests that some or all of these activities are to be assisted, or entirely carried out, autonomously by the software itself, rather than being done solely by humans. There seems to be interesting research required in order to reduce the dependence upon humans in most of these activities. In this paper we describe how we propose to transition much of the responsibility for testing and evaluation of software from humans and onto automated tools and processes.

The essence of our idea is that deployed software should be under continuous testing, analysis, and evaluation, drawing heavily upon the computational resources and actual utilization patterns found in the deployment environment. As the amount and nature of possible testing and analysis are virtually limitless, there is value in continuing these activities perpetually. It is important, however, that the activities not be unfocussed or undirected. We propose that the process of perpetual testing and analysis be guided by incremental results and findings so as to produce increasingly sharp and definitive results. It is suggested that these results will inevitably lead to increasingly well-informed suggestions about what modifications to the software seem most likely to lead to real improvements. Thus, our perpetual testing and analysis proposal is aimed not only at

reducing the need for human involvement in evaluation, but also at assisting human effort in proposing improvements.

There are substantial technical challenges in doing this. We propose that both dynamic testing and static analysis go on essentially endlessly, but that results from each of these activities be used to focus and sharpen the other's activity. Research into how to best make these two complementary evaluation approaches most synergistic is needed. We envision orchestrating this synergy through the use of precisely defined processes, and there is considerable research to be done in this area as well.

2.0 APPROACH

While self-modifying code has existed since the earliest days of computing, there is ample evidence that it poses clear dangers. Of most concern to us is the fact that self-modifying code is generally difficult or impossible to analyze, and therefore the range of its possible behaviors is difficult or impossible to bound. For this reason we seek a way in which software can modify itself, without necessitating the need for code to modify itself. Our approach to this problem is to view a software product as an aggregate of diverse types of software artifacts, including such types of components as a requirements specification, an architecture, low level design specifications, code, test cases, and analysis results. We also suggest that the software product also include as a component the process by which various components of the software product (code, in particular) are to be modified. By doing so we can assure that no component of the software product need modify itself. Rather, a separate component, the modification process, is responsible for modifying other components, such as the code.

Going further, we suggest that the software product is also characterized by a variety of constraints that specify the way in which the various components should be related to each other. The constraints are particularly important because they are to be used to determine when and how the components of the product need to be modified. Thus, for example, when test results derived through execution of testcases are inconsistent with the requirements to which they should be related, this is a signal that product modification is needed.

The foregoing suggests that a collection of tools should also be considered to be part of the software product. These tools are the devices that are used to help build the components of the product and to determine the degree to which the product is internally consistent. Thus compilers, design tools, testcase monitors, and static analyzers are all examples of tools that should be considered to be part of the overall software product.

In classical software development, the tools are applied in order to build and test the product and its components right up until deployment. But at deployment the product code is separated from the rest of the components, and from its constraints and tools, and is placed in the deployed environment. This complicates the modification of the code substantially.

We propose that deployed software product code remain tethered to the other product components (including the software product modification process), as well as to the constraint and tool sets that comprise a complete product. By doing this, it becomes possible for tools to continue to evaluate the consistency of the code with other artifacts and to effect modifications. This is the more precise sense in which our proposal suggests how to effect the self-modification of software, without necessitating the self-modification of the code component.

Thus the perpetual testing approach implies that software code be perpetually enhanced by access to an environment that supports its evaluation and improvement (we shall refer to this as the development environment, even though it will persist past initial development), and takes a pro-active role in assuring that evaluation continues to make positive contributions to the improvement of the software. Coordination of diverse and numerous types of testing and analysis artifacts that exist in the development and deployed environments is a daunting job that is prohibitively expensive and error-prone if carried out manually. Instead, a highly automated testing and analysis process integrates testing and analysis tools and artifacts, using the available computing resources that can be acquired at any given time. Although this process is considered to be an integral part of the product, it is not required to be resident with the code in the deployed environment.

3.0 TECHNICAL AND RESEARCH CHALLENGES

3.1 DEPLOYMENT CONSIDERATIONS

It is important to emphasize that the nature of the interconnection and interaction between the development environment and the deployed code that is (presumably) being testing perpetually will vary considerably, depending upon the considerable variation in deployment situations. Thus, for example, it may be quite reasonable to suggest that non-critical prototype research software deployed in a university research setting may be under continuous evaluation, and continual interaction with its development environment. On the other hand, it is unthinkable to suggest that mission critical realtime software code deployed in a secure military environment will have direct contact with its original development environment while it is deployed and in service. There is a large spectrum of deployment situations between these two extreme situations. It is our belief, however, that we can fashion a corresponding spectrum of approaches to supporting the provision of the benefits of perpetual testing. Certainly the degree to which it is feasible to connect deployed code to the rest of the product will dictate the degree of success we are likely to have with self-evaluation and self-improvement.

Clearly the furnishing of parameterized monitoring capabilities, controls over communication between deployment and development environments, and rehosting of major portions of the development environment all pose significant technical challenges.

3.2 ANALYSIS AND TESTING RESEARCH

3.2.1 INCREMENTAL RETESTING AND REANALYSIS

Testing and analysis techniques should be applicable continuously during initial software development, where it should be applied to incomplete systems, and continuously as software modifications are considered and carried out. As modifications are being considered, it is highly desirable not to repeat analysis and testing for unchanged aspects of the software product. This entails the ability to carry out careful analysis of the impacts of proposed modifications, determination of which past analytic results remain valid, determination of the most pressing retesting/reanalysis needs, and an optimized process for addressing the most pressing needs first.

Incremental reanalysis and testing will require technologies for creating a rich web of summary information about the code and its various subcomponents, deciding what parts are partly or fully reusable, and optimizing updates to portions that changes have rendered invalid. Incremental approaches have been developed for compilation, based on interprocedural data-flow analysis techniques. Incremental techniques developed for testing and analysis will be conceptually similar, but will involve a much more complex set of constraints and relations among multiple sources and kinds of information. The nature of these relations is indicated briefly below.

3.2.2 INTEGRATION OF TECHNIQUES

The prospect of Perpetual Testing suggests that diverse analysis and testing techniques will have to be synergistically integrated and reintegrated dynamically and in diverse ways. Previous research has described how different analysis techniques could complement one another, but have proposed fixed integration strategies consistent with the assumption that testing was a phase of fixed duration. These activities have teamed less expensive, but usually less precise, analysis techniques with more expensive, but very precise, techniques, often greatly reducing computation costs and human intervention. For example, static dataflow analysis scans can focus and sharpen dynamic testing regimens, and can be used to iteratively generate and solve successions of dataflow analyses, that iteratively sharpen analytic results. These prototype integrations indicate many opportunities for synergy, and the clear feasibility of effecting new integrations dynamically.

Verification systems have traditionally maintained constraint relations among lemmas and theorems, but have had an overly simplistic view that a property is either proved (when all its relations are proved), or is completely without support. Testing systems have maintained records of thoroughness (various notions of coverage), as well as sets of properties in the form of test oracles, but have not related levels of assurance to particular properties. Static analysis tools have been limited to a distinction between verified properties (“must” results) and inconclusive (“may”) results.

Integration of analysis and testing techniques, and particularly integration of post-deployment usage information, provides an opportunity for a far richer set of constraints among properties, techniques, and levels of assurance. For

example, a property may be “”verified” by a static dataflow analysis, dependent on the absence of an aliasing relation that is monitored in testing. Each asserted property of a software system can be supported by a collection of analyses and assurances of differing strengths, which are propagated through the web of relations and constraints. Monitoring of deployed software is critical in this regard, and explicit constraint links on patterns monitored in the deployed software provide a way to calibrate and strengthen assurances established in the development environment.

3.2.3 SPECIFICATION-BASED ANALYSIS

Testing is a human intensive process. Testers must develop test cases, run the test cases and then evaluate the results. The latter task can be particularly time consuming, tedious, and error prone. Specification-based testing techniques are being developed so that humans no longer have to play the role of oracle. The use of specifications needs to be further developed, not only for testing but for a range of analysis techniques.

Specifications are the driving impetus for a number of analysis approaches. Some modern dataflow analyzers use a quantified form of regular expressions for expressing properties that are to be validated, others use temporal logic specifications, while still others use graphical interval logic. The processing that is actually done and the validity of the system being evaluated depends on the specification. Needless to say the quality of the analysis also depends on the quality of the specification. For example, data flow analysis will not determine the validity of a property unless that property is captured by a specification.

An advantage of specification-based analysis and testing is that the specification base can grow. Over time, as users have more experience with the software product, new specifications can be formulated. Every fault that is discovered after deployment should be captured by a new specification of the intended behavior so that future modifications of the software product can be evaluated against this specification as well. As the specification base grows, more and more of the reanalysis and retesting will be driven by this base. This should improve the quality of the testing and analysis, but should also improve the quality of the software product, reduce the amount of human intervention, and greatly reduce the time devoted to testing and analysis.

3.2.4 PREDICTIVE-DRIVEN ANALYSIS AND TESTING

A perpetual testing framework provides a unique opportunity to gather metrics about the sequence of modifications to the software product and to use those metrics to predict the most appropriate analysis or testing technique for evaluation of subsequent modifications. For example, a code subcomponent that has a history of containing faults might be given a high priority for reanalysis if it is modified. Information about the kinds of faults that were discovered in the past and metrics about the component itself, such as if it uses concurrency or has complicated data structures, would impact the choice

of analysis approach to employ. Also, past execution costs could be used to predict computing resources that would be needed to complete the analysis in a timely fashion. Finally, if one technique proved to require substantial human interaction in the past for problems with a similar metric footprint, then less consumptive techniques would be considered.

Recent work on testing and analysis has just started to address using metrics to drive the choice of analysis techniques. With perpetual testing we expect to be able to be more complete and effective in gathering information about the software being evaluated, the kinds of faults discovered (or properties verified) by the analysis and testing techniques, the computing resources used, and the amount of human intervention required. On the basis of this information, we should be able to develop a predictive model of what appears to be the best testing and analysis process. This model would itself be the subject of evaluation and would continue to be modified as we gathered more experimental evidence. This should lead to a predictive meta-model to be incorporated as part of the overall software product. As more information is learned about a particular software product and its modification history, this information would be used by the meta model to evolve a predictive model to drive the testing and analysis process for that software.

3.2.5 PERPETUAL TESTING PROCESS DEVELOPMENT

Analysis and testing activities should be viewed as processes to be developed as software is, from requirements through coding, evaluation, and evolution. This is particularly important for perpetual testing processes. Because they are to be indefinitely ongoing processes, it is essential that clear goals be specified beforehand, so that progress towards those goals be continually monitored, and so that revisions to either goals, of processes, or both can be made continually. Thus, perpetual testing can reasonably be viewed as the integration of sequences of applications of both testing and analysis tools in demonstrable support of precisely articulated analysis and testing goals. We propose to demonstrate the value of articulating requirements for perpetual testing processes, and then also demonstrate perpetual testing process architectural designs.

It seems particularly appropriate to consider, in addition, the development of code for analysis and testing processes. For example, dynamic regression testing entails iteration that is comfortably expressed with traditional loops. Deciding when and whether to follow coarse-grained, static dataflow analysis with more precise, sharpened dataflow analyses is expressible with traditional case and if-then-else constructs. In both cases, the fine scale detailed analysis products must also be specified.

The testing and analysis process we envisage will also require reactive control mechanisms. It is often important to program immediate reactive responses to such testing failures as incorrect results and system crashes. Dataflow analysis results may need to trigger fixed standard subsequent reanalysis. The

reporting of analysis results obtained at deployment sites might be triggered by timer events, or by the very action of their being completed.

All of this argues for the specification of actual executable process code to guide the execution of these perpetual testing processes, and therefore the continuous self-evaluation of software products. In our work we have demonstrated such perpetual testing process code, and now propose that it be used as the engine for driving software product self-improvement.

4.0 SUMMARY

The perpetual testing and analysis approach promises to enable the continuous self-evaluation of software throughout the sequence of modifications occurring during its entire lifetime, and thereby to enable software self-improvement that can be measured and evaluated. Key to doing this is to perpetually link deployed software code to the rest of the overall software product's components, constraints, tools, and the perpetual testing process itself. This should ultimately increase the confidence that people will have in their software products. As software products become larger and more complex, they will become ever more difficult to evaluate and improve, trust, and predict unless an approach such as perpetual testing is explored.

5.0 ACKNOWLEDGEMENTS

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, and by U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, the U.S. Government, the National Science Foundation, or of IBM.